

Sc N 85 / 118 A

UTILISATION DES SYSTEMES DE REECRITURE D'EXPRESSIONS FONCTIONNELLES COMME OUTILS DE TRANSFORMATION DE PROGRAMMES ITERATIFS

THESE DE DOCTORAT D'ETAT EN INFORMATIQUE
PRESENTEE PAR



Françoise Bellegarde

SOUTENUE LE 22 JUIN 1985

DEVANT LA COMMISSION D'EXAMEN

PRESIDENT
RAPPORTEURS

C. PAIR
G. BERRY
P. LESCANNE
J.F. PERROT
J.P. FINANCE
J. HSIANG
J.P. JOUANNAUD
D. TURNER

EXAMINATEURS

BIBLIOTHEQUE SCIENCES NANCY 1



D

095 147877 8

UTILISATION DES SYSTEMES DE REECRITURE
D'EXPRESSIONS FONCTIONNELLES
COMME OUTILS DE TRANSFORMATION
DE PROGRAMMES ITERATIFS

THESE DE DOCTORAT D'ETAT EN INFORMATIQUE
PRESENTEE PAR

Françoise Bellegarde

SOUTENUE LE 22 JUIN 1985

DEVANT LA COMMISSION D'EXAMEN

PRESIDENT
RAPPORTEURS

EXAMINATEURS

C. PAIR
G. BERRY
P. LESCANNE
J.F. PERROT
J.P. FINANCE
J. HSIANG
J.P. JOUANNAUD
D. TURNER

Under no illusions he'll make his dream come true
Calculating long shots
Still surprised to find something new

C'est sans illusions qu'il va réaliser son REVE
Calculant ses risques
Encore surpris de trouver du nouveau

MURRAY HEAD

Je veux remercier ici:

Claude Pair qui me fait l'honneur de présider ce jury de thèse. C'est lui qui, par le sujet qu'il m'a suggéré aux origines de cette thèse et par ses idées sur les méthodes de construction des programmes itératifs, a suscité ce travail.

Pierre Lescanne qui a toujours été prêt à proposer des idées et des lectures, a toujours su critiquer et encourager. Il a accepté de diriger mes recherches. Je lui suis doublement reconnaissante: sans REVE, dont il est l'inventeur, ce travail n'aurait pas existé.

Jean-Pierre Jouannaud qui a également dirigé ce travail. Je lui suis énormément reconnaissante d'avoir créé l'environnement scientifique sans lequel, je ne pouvais pas approfondir mes idées et trouver des solutions aux problèmes que je me posais. C'est lui qui m'a apporté la connaissance dans le domaine de la réécriture.

Gerard Berry a accepté la tâche de rapporteur, ce qui est pour moi un très grand honneur.

Jean-François Perrot, à l'occasion d'un exposé sur mon travail, a su me conseiller et surtout m'encourager, ce qui est sans doute le plus précieux.

Jieh Hsiang, pour les discussions intéressantes que nous avons eues, mais surtout pour son écoute et sa gentillesse durant son séjour au sein de l'équipe Eureka.

David Turner me fait l'honneur et le plaisir de participer à ce jury. Je lui suis reconnaissante de l'intérêt qu'il a porté à mon travail et d'avoir eu la patience de lire cette thèse en français.

Je remercie spécialement Jean-Luc Remy qui a eu l'extrême gentillesse de lire et de corriger les versions successives de cette thèse, envoyant encore corrections et remarques depuis les USA. Il m'a, depuis toujours, encouragée et aidée dans ce travail.

Je dois ajouter que je dois beaucoup à tous les membres de l'équipe EURECA. Une mention spéciale revient à Claude et Hélène Kirchner pour toute l'aide apportée au cours de l'élaboration commune de nos thèses.

Un grand merci aussi à André et Anne qui n'ont ménagé ni leur peine, ni leur bonne humeur pour m'aider.

Plan de la première partie

INTRODUCTION

PROGRAMMATION FONCTIONNELLE AVEC DES SEQUENCES	17
1. Introduction	17
1.1. Programmation fonctionnelle	17
1.2. Programmation avec des séquences	18
1.3. Programmation structurée	22
1.4. Contenu du chapitre	24
2. Un langage pour exprimer et transformer les programmes itératifs	25
2.1. Les objets	26
2.2. Les fonctions	27
3. Les primitives de base	28
3.1. Les primitives sur les séquences	31
3.2. Les formes fonctionnelles de base	35
4. Construction d'expressions fonctionnelles simples	49
4.1. Les séquences intermédiaires sont issues des compositions d'itérations	51
4.2. Quand l'utilisation d'expressions connues ne donne pas une bonne solution	59
4.3. A propos du raisonnement descendant	67
4.4. A propos des conditionnelles	75
4.5. Exemple 10. Il faut connaître des propriétés des fonctions qu'on utilise	78
4.6. Un dernier exemple	79

5. Les expressions fonctionnelles	85
5.1. Expression fonctionnelle	86
5.2 Les fonctions sont toutes strictes	88
5.3. Selecteurs et illisibilité des expressions FP	90
6. Bibliographie	
THEORIE EQUATIONNELLE VALIDEE PAR L'ALGEBRE DES EXPRESSIONS FONC- TIONNELLES	95
1. L'algèbre des expressions fonctionnelles	96
1.1. Définition	96
1.2. Les FP-termes	98
1.3. Opérations sur les termes : remplacement et substitution	
2 Equations valides dans la FP-algèbre	104
2.1. Les relations d'équivalences	104
2.2. Equations valides dans la FP-algèbre	106
2.3. Equations conditionnelles valides dans la FP-algèbre	114
2.4. Algèbre et théorie équationnelle	117
3. Axiomes d'une théorie équationnelle	124
3.1. Axiomes de base	125
3.2. Des axiomes sur "ALL"	128
3.3. Axiomes sur "*"	130
3.4. Axiomes sur "WH"	135
3.5. Axiomes pour les primitives sur les séquences	138
3.6. Axiomes sur "->_;"	148
4. Théories particulières	152
4.1. Première règle	153
4.2. Deuxième règle	157
4.3. Troisième règle	162

4.4. Conclusion	167
5. Bibliographie	
UNE ANALYSE D'EXPERIENCES UTILISANT REVE	171
1. Systèmes de réécriture de termes	171
1.1. définitions	171
1.2. Terminaison d'un système de réécriture	173
1.3. Confluence	181
1.4. Algorithme de Knuth-Bendix	182
2. Appliquer REVE sur les axiomes de la FP-algèbre	187
2.1. Les particularités de REVE	187
2.2. Utiliser REVE sur l'ensemble d'équations défini au chapitre II	191
2.3. Conclusion: notre méthode est expérimentale	201
3. L'expérimentation	202
3.1. Les axiomes de base	203
3.2. Les axiomes sur les primitives	207
3.3. Les axiomes sur "ALL"	229
3.4. Les axiomes sur "*"	237
3.5. Les axiomes sur "WH"	257
3.6. Les axiomes sur "->_;"	263
4. Utilisation des systèmes de réécriture trouvés	273
4.1. Utilisation de S-BASE	273
4.2. Utilisation de S-PRIMITIVE	274
4.3. Utilisation de S-ALL	276
4.4. Utilisation de S-* (ou S-WH) et de S-** (ou S-WHHH)	280
4.5. Rôle des système S-CONDO, S-COND1 et S-COND2	282
5. Conclusion	282

INTRODUCTION

Motivations

Quand on construit un programme à l'aide d'un raisonnement structuré (descendant ou ascendant), on introduit des intermédiaires inutiles. Je l'ai remarqué plus particulièrement lors de séances d'initiation à la programmation, qui utilisaient la méthode de construction de programmes MEDEE ([Pai79],[Duc84],[Fin85]). Cette méthode guide ses utilisateurs vers la construction structurée de programmes itératifs en utilisant le langage MEDEE qui est applicatif [Tur81] et dont les objets sont essentiellement des suites comme dans le langage LUCID [Ash76].

Nous référant à MEDEE, nous considérons que le résultat d'une itération est une séquence (ou suite). Aussi quand on construit un programme itératif structuré, on introduit naturellement des séquences intermédiaires inutiles. Par exemple le produit scalaire de deux séquences

$$\langle x_1, x_2, \dots, x_n \rangle \text{ et } \langle y_1, y_2, \dots, y_n \rangle$$

est obtenu en construisant la séquence

$$S_1: \langle x_1 y_1, x_2 y_2, \dots, x_n y_n \rangle$$

puis les cumuls

$$S_2: \langle 0, x_1 y_1, x_1 y_1 + x_2 y_2, \dots, x_1 y_1 + x_2 y_2 + \dots + x_n y_n \rangle$$

dont le produit scalaire est le dernier terme. La séquence S_1 est inutile. Notre but est de transformer les programmes itératifs pour en ôter les séquences intermédiaires inutiles.

On peut évidemment résoudre le problème par des optimisations au moment de la compilation. Des travaux sur la compilation des programmes APL [Gui78] font ces optimisations. On peut aussi exécuter avec une méthode de "lazy evaluation" [Hen76], qui n'évalue pas les séquences intermédiaires inutiles.

Ces solutions ne nous conviennent pas. Le langage MEDEE n'est pas appelé à être utilisé comme langage de programmation, mais à être traduit dans des langages algorithmiques classiques comme PASCAL, voire PL1 ou BASIC. Nous voulons plutôt étudier les transformations à effectuer sur le programme source. Nous voulons que cette étude montre comment améliorer l'enseignement des méthodes de construction structurées des programmes. Un bon moyen peut être de montrer directement les transformations effectuées sur le programme par un système de transformation.

Il nous suffit donc de définir un système de transformation permettant de supprimer les séquences intermédiaires inutiles des programmes itératifs.

Choix d'un langage fonctionnel pour coder les programmes

Notre idée de base est assez naive. Si l'on regarde un programme P d'un point de vue fonctionnel, le résultat r est le résultat de l'application de P (qui est alors une expression fonctionnelle) à ses données d.

$$r = P(d)$$

Si x est un résultat intermédiaire, il existe une expression fonctionnelle P1 telle que

première partie

$$x = P1(d)$$

et une expression fonctionnelle P2 telle que

$$r = P2(x).$$

P est alors l'expression de la composition de P1 et P2, et si on note "o" la composition des fonctions, l'expression de P est

$$P = P2 \circ P1.$$

Si x est un intermédiaire inutile, il doit exister une autre expression de la fonction P qui ne contient plus cette occurrence de "o".

Il est alors facile de regarder les programmes d'un point de vue algébrique en utilisant l'algèbre des fonctions qui peut servir de base d'axiomes pour nos transformations.

Nous avons signalé que MEDEE est un langage applicatif. Il est assez facile de coder les programmes MEDEE dans un langage fonctionnel, cela permet de se débarrasser du "sucré syntaxique" et des redondances qui figurent en MEDEE et qui proviennent du fait que le langage sert de guide à une méthode de construction du programme.

Utiliser un langage fonctionnel existant est plus judicieux si l'on veut communiquer facilement les résultats de son travail. Parmi les langages fonctionnels, c'est le langage FP [Bac78] qui nous a paru le mieux adapté.

Nous allons expliquer rapidement la raison principale de notre choix.

Pour coder un programme itératif dans un langage fonctionnel, il suffit de formuler les itérations par des fonctionnelles appropriées. Une

première partie

fonctionnelle s'applique à une fonction et a pour résultat une autre fonction. En FP, on trouve par exemple la fonctionnelle "ALL" qui est aussi "mapcar" en LISP [McC62]. Soit une fonction f , "ALL f " est une fonction qui, appliquée à une séquence $\langle x_1, \dots, x_n \rangle$, a pour résultat la séquence $\langle f(x_1), \dots, f(x_n) \rangle$. "ALL f " code une itération.

Le langage FP nous permet de définir un petit ensemble de fonctionnelles pour coder les itérations.

Un langage fonctionnel peut être un modèle du LAMBDA-CALCUL ([Ber81a], [Cou85]). En général, les fonctions sont exprimées par leur application à des variables qui sont à l'origine de l' α -conversion. Mais en FP, l'utilisation des fonctionnelles "o" pour la composition des fonctions et "[_]" pour les n-uplets de fonctions, des sélecteurs pour les projections permet la suppression des variables. Grâce à ces fonctionnelles, on a une algèbre libre des fonctions. Si on ajoute les fonctionnelles permettant de coder les itérations, toute fonction codant un programme itératif s'exprime en FP comme un terme de cette algèbre libre. Considérer les programmes et leur transformation d'un point de vue algébrique est alors évident.

La FP-algèbre

Maintenant, un programme itératif est une expression fonctionnelle FP. Nous considérons un ensemble de variables de fonctions V et nous voyons les expressions FP comme des termes (nous dirons les FP-termes) d'une algèbre libre dont les symboles fonctionnels sont les fonctionnelles du langage FP et les symboles des constantes sont ceux des fonctions primitives FP et les notations des fonctions constantes.

première partie

L'égalité que nous voulons décrire dans la FP-algèbre est l'égalité extensionnelle

$$f = g \text{ si et seulement si } f(x) = g(x) \text{ pour toute donnée } x.$$

Cette égalité nous permet de valider un ensemble A d'équations de la FP-algèbre. Ces équations nous servent d'axiomes.

Nous avons par exemple l'axiome d'associativité de la composition des fonctions

$$(f \circ g) \circ h = f \circ (g \circ h)$$

mais aussi des axiomes sur "ALL" comme

$$\text{ALL } f \circ \text{ALL } g = \text{ALL}(f \circ g)$$

L'ensemble des axiomes A nous permet de définir une A -égalité. Si nous incluons dans l'ensemble d'axiomes toutes les manières de composer les fonctions ayant pour argument et pour résultat des séquences, chaque classe de A -égalité contient un terme qui n'engendre pas de séquence intermédiaire inutile.

Pour l'axiome cité ci-dessus, la séquence générée par "ALL g " dans le premier membre n'existe plus dans le second membre.

On peut montrer simplement par remplacement d'égaux par égaux que deux expressions fonctionnelles sont A -égales. Notre but est de trouver un des termes de la classe de A -égalité qui n'engendre pas de séquence intermédiaire inutile.

Les systèmes de réécriture

première partie

Nous pouvons orienter les axiomes en règles d'un système de réécriture de manière à ce que la relation de réduction supprime les séquences intermédiaires inutiles. Nous aurons par exemple la règle

$$\text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$$

Si, comme nous l'avons déjà dit plus haut, nous parvenons à inclure dans le système de réécriture les axiomes qui concernent toutes les compositions des fonctions ayant pour résultat une séquence avec les fonctions ayant pour argument une séquence, la forme normale d'un FP-terme, si elle existe, n'engendre plus de séquence intermédiaire inutile.

L'idéal est que tout terme ait une forme normale unique. Mais il faut pour cela que le système de réécriture obtenu soit noethérien et confluent c'est à dire convergent [Hue80a].

La procédure de Knuth-Bendix [Knu70] permet de transformer un ensemble d'équations en un système de réécriture convergent équivalent. "équivalent" signifie que deux termes A-égaux ont une même forme normale.

Selon un ordre bien fondé sur les termes, la procédure de Knuth-Bendix oriente les équations en règles de réécriture de manière à ce que le système soit noethérien. Elle complète le système de réécriture avec des conséquences équationnelles de l'ensemble des axiomes pour rendre le système de réécriture localement confluent. Si la procédure n'échoue pas (pour orienter une règle) et si elle ne diverge pas (en engendrant indéfiniment de nouvelles règles pour compléter le système), elle a pour résultat un système de réécriture convergent.

Nous avons donc défini un ensemble d'équations de la FP-algèbre qui

contient toutes les manières de composer les fonctions codant les itérations. Puis nous avons utilisé la procédure de Knuth-Bendix implémentée dans le système REVE [Les83] pour essayer d'obtenir un système de réécriture convergent à partir de ces axiomes. Ce faisant, nous nous sommes heurtés à de nombreux problèmes. Il faut noter que c'est certainement la première fois que l'on utilise une implémentation de la procédure de Knuth-Bendix pour un aussi grand nombre d'axiomes.

Les résultats

Nous avons isolé des systèmes de réécriture convergents qui sont équivalents à des sous-ensembles d'axiomes de la FP-algèbre. REVE nous permet d'obtenir la forme normale $t \downarrow_R$ d'un FP-terme t pour un de ces systèmes de réécriture R . $t \downarrow_R$ engendre moins de séquences intermédiaires que t et nous pouvons dire que nos systèmes sont capables d'éliminer les séquences intermédiaires dues aux compositions des fonctions qui codent les itérations.

Les problèmes

Cependant nous ne sommes pas tout à fait satisfaits par ces résultats.

Si nous voulions vraiment minimiser le nombre de séquences intermédiaires engendrées, il faudrait obtenir un système de réécriture équivalent à la FP-algèbre.

Il est impossible d'obtenir un système d'axiomes complet qui décrive l'égalité extensionnelle. Mais il sera possible d'obtenir de meilleurs résultats quand nous aurons à notre disposition des outils de réécriture plus performants.

1) Preuve de terminaison

REVE oriente les équations en règles en utilisant l'ordre dit de "recursive decomposition" ([Les81],[Jou81b],[Jou82b]). Cet ordre ne nous permet pas dans tous les cas d'orienter les axiomes dans le sens qui nous convienne, c'est à dire tel que la réécriture d'un terme par la règle correspondante élimine des séquences intermédiaires. Par exemple, au cours de notre expérimentation, l'équation

$$\text{ALL } f \circ (\text{ALL } g \circ h) = \text{ALL}(f \circ g) \circ h$$

est orientée de droite à gauche par REVE alors que c'est l'autre orientation, de gauche à droite, qui réécrit un terme en éliminant la séquence engendrée par "ALL g".

Nous ne devons donc utiliser du système REVE que la partie algorithme de complétion et orienter les règles "manuellement". La preuve de terminaison des systèmes est faite, ensuite, par d'autres méthodes [Lan79]. Mais cette preuve devient très vite insurmontable sans assistance de calcul. Il faudrait trouver et implémenter dans REVE des méthodes de preuves "plus complètes", ce qui est prévu ([Gna85],[Alh85]).

2) Quand la procédure de Knuth-Bendix diverge,

parce qu'elle engendre indéfiniment de nouvelles règles pour compléter le système. On peut capter l'ensemble infini de règles par des méta-règles ([Kir82],[Kir85b]). Il faut aussi savoir compléter un système de réécriture comportant des méta-règles et étendre la réécriture à ces systèmes [Kir85b].

3) Un nombre infini d'axiomes dans la FP-algèbre

Les systèmes de réécriture ne peuvent inclure qu'un nombre fini d'axiomes. La FP-théorie en contient un nombre infini, parce que les sélecteurs sont en nombre infini et que la fonctionnelle "[__]" est d'arité multiple. On a par exemple une famille infinie d'axiomes

$$[f_1, f_2, \dots, f_n] \circ h = [f_1 \circ h, f_2 \circ h, \dots, f_n \circ h]$$

Il est possible de coder ces familles infinies d'axiomes par des schémas d'axiomes. Il est aussi possible d'étendre la notion de réécriture pour inclure des schémas de règles. Il est facile de voir que les résultats de convergence obtenus pour les systèmes qui comprennent un des axiomes de la famille, par exemple

$$[f_1, f_2] \circ h = [f_1 \circ h, f_2 \circ h]$$

se généralisent avec les schémas de règles.

Il n'existe pas, pour le moment, de généralisation de la procédure de Knuth-Bendix qui accepte des schémas d'axiome. Mais cette possibilité peut être envisagée en liaison avec une version ultérieure du système REVE qui accepte les méta-règles [Kir85b].

4) La FP-algèbre contient des axiomes permutatifs

Un axiome permutatif ne peut pas devenir une règle d'un système de réécriture. On doit réécrire modulo ces axiomes ([Jou84a],[Der83]). Pour inclure les axiomes permutatifs de notre FP-algèbre, il faudra utiliser la version REVE 2.3 qui doit être prochainement opérationnelle ([Kir84],[Kir85a],[Kir85b]).

5) La FP-algèbre contient des axiomes conditionnels

Dans la version utilisée, REVE n'accepte que des théories "purement" équationnelles. On ne peut inclure d'axiomes "conditionnels". Ces axiomes donnent en effet des règles de réécriture conditionnelles de la forme

$$g \Rightarrow d \quad \text{SI } P$$

P est une formule booléenne qui doit être prouvée vraie pour que l'on ait $g = d$. On impose toujours ([Rem82],[Kap84]) que les variables de P soient incluses dans celles de g.

On doit évaluer récursivement les prémisses P avant d'appliquer la règle $g \Rightarrow d$. La FP-algèbre contient des axiomes conditionnels. Pour construire nos systèmes de réécriture, nous parvenons à éviter le problème par des stratégies appropriées d'utilisation de REVE. Nous verrons que la plupart des prémisses se ramènent à un problème de contrôle de type. Il faudra donc typer les expressions fonctionnelles pour pouvoir effectuer ce contrôle. Pour les axiomes conditionnels, il restera à exploiter les possibilités qui seront offertes par des versions ultérieures de REVE, par exemple la version REVEUR 4 [Rem84].

6) Pour améliorer les programmes itératifs,

nous nous limitons au problème des séquences intermédiaires inutiles. Nous nous contentons de considérer les axiomes des primitives concernant les séquences. Mais les programmes utilisent d'autres primitives (des additions de deux entiers, des comparaisons entre entiers,...) et si nous voulons aller plus loin, il faut inclure les axiomes de ces primitives.

première partie

Nous voyons qu'il faut alors aussi tenir compte de ce que nous appellerons des règles d'inférence qui peuvent se mettre sous la forme

$$u = v \quad | - \quad g \Rightarrow d$$

L'évaluation de la prémisse $u = v$ est une décision d'égalité et on se ramène à une réécriture conditionnelle à condition que les variables de u et de v soient incluses dans g [Kap84]. Mais c'est un peu plus compliqué car il existe dans u et v des variables qui ne sont ni dans g ni dans d et il s'agit bien de décider l'égalité $u = v$ avant d'appliquer $g \Rightarrow d$.

Le problème est encore plus complexe quand il existe dans u ou v des variables de d qui ne figurent pas dans g car alors, évaluer la prémisse demande de résoudre l'équation $u = v$.

Conclusion

Le plan de la thèse suit celui de l'introduction. Une première partie est consacrée à l'exposé des résultats acquis en utilisant la version de REVE dite 2.2. Une deuxième partie permet d'envisager un plan de ce que l'on pourra sans doute faire avec les versions ultérieures de REVE.

1) Dans la première partie,

le chapitre I est en quelque sorte une vaste introduction exposant les motivations du travail, donnant des exemples montrant comment la construction des programmes introduit naturellement des séquences intermédiaires inutiles et justifiant les choix. En particulier, on justifie le choix des fonctionnelles introduites dans le langage FP pour coder les itérations.

Le chapitre II est consacré à la définition des axiomes de la FP-

Première partie

algèbre. Il définit aussi quelques règles d'inférences utiles pour transformer certains cas particuliers posés par nos exemples de programmes.

Le chapitre III est le chapitre clé de la thèse. Il donne un compte rendu détaillé de l'expérimentation conduite avec la version de REVE 2.2 pour définir des systèmes de réécriture convergents à partir des axiomes de la FP-algèbre. On voit aussi quel effet ont ces systèmes de réécriture sur les expressions fonctionnelles qui codent les programmes.

2) Dans la deuxième partie,

Le chapitre I prouve la terminaison des systèmes de réécriture, preuve qui n'a pas pu être faite par le système REVE.

Le chapitre II concerne l'étude des divergences de REVE et envisage de la possibilité de les résoudre en utilisant des méta-règles.

Le chapitre III montre la possibilité de généraliser les systèmes de réécriture définis dans la première partie aux infinités d'axiomes de la FP-algèbre qui concernent la fonctionnelle $[__]$ et les sélecteurs par utilisation de schémas de règles.

Le chapitre IV montre pourquoi il faut typer les expressions fonctionnelles, pose le problème des axiomes conditionnels et des règles d'inférence.

1. Introduction

1.1. Programmation fonctionnelle

Nous pouvons considérer que le résultat r d'un programme ayant pour entrée une donnée d est le résultat d'une application " $P(d)$ ". Ce point de vue conduit à une expression dite "applicative" du programme. On peut structurer la formulation du programme en introduisant des "résultats intermédiaires" et si un seul "résultat intermédiaire" x est introduit et se nomme x , le programme s'énonce par deux expressions

$$x = P_1(d) \text{ et } r = P_2(x).$$

Il est alors possible de considérer le programme comme une fonction P sans nommer ni r , ni x , ni la donnée d . P est défini par l'expression fonctionnelle " $P_2 \circ P_1$ ".

Structurer P revient alors à trouver des fonctions $P_1, P_2, \dots, P_n$, telles que $P = P_n \circ \dots \circ P_2 \circ P_1$. P est décrit par une expression fonctionnelle. Le symbole de composition de deux fonctions que l'on note " $\circ$ " est ce qu'on appelle une "forme fonctionnelle". Une forme fonctionnelle s'applique à des fonctions et a pour résultat une fonction. Citons aussi la forme fonctionnelle "ALL" qui est le "mapcar" de LISP. "ALL" s'applique à une fonction f et a pour résultat une fonction "ALL f ". Cette fonction "ALL f ", appliquée à une séquence $\langle x_1, x_2, \dots, x_n \rangle$ a pour résultat une autre séquence $\langle f(x_1), f(x_2), \dots, f(x_n) \rangle$.

Une conséquence est que l'on possède alors une algèbre des expressions fonctionnelles. Un axiome bien connu de cette algèbre est celui de l'associativité de la composition de deux fonctions

$$(f \circ g) \circ h = f \circ (g \circ h)$$

Les preuves d'équivalence entre programmes sont alors des preuves d'équivalence entre leurs expressions fonctionnelles dans cette algèbre.

C'est l'élégance et une certaine simplicité des preuves d'équivalence de programmes qui nous ont conduit à nous intéresser aux programmes écrits sous forme d'expressions fonctionnelles. Après d'autres tentatives [Bel80], nous avons choisi le langage fonctionnel FP ([Bac78],[Bac81a],[Bac81b]) comme langage d'expression des programmes.

1.2. Programmation avec des séquences

Nous appelons séquence une suite finie. Toute itération qui se termine, dans un programme, sous-entend le calcul d'une séquence. Considérons un programme exprimé en PASCAL

```
PROGRAM PUIS;
VAR
  P,I:INTEGER;
BEGIN
  P:=1; FOR I:=1 TO 5 DO P:=P*2;
  WRITE(P)
END.
```

L'itération affecte à P la séquence <1,2,4,8,16,32> pour en écrire le dernier terme 32. Nous pouvons toujours considérer qu'une itération qui se termine a pour résultat une séquence dont on n'utilise souvent que le

première partie

dernier terme.

En programmation fonctionnelle nous trouverons des formes fonctionnelles pour construire des fonctions ayant pour résultat des séquences. "ALL", déjà citée, est un exemple d'une telle forme fonctionnelle. Dans l'expression fonctionnelle, les itérations sont exprimées par les fonctions (construites avec des formes fonctionnelles) ayant pour résultat des séquences. Des fonctions primitives sur les séquences seront utilisables. Une primitive "last" appliquée à une séquence $\langle x_1, x_2, \dots, x_n \rangle$ donne le dernier terme x_n .

Considérer qu'une itération a pour résultat une séquence plutôt que son dernier terme permet de composer deux itérations. La séquence s résultat d'une itération I peut être argument d'une autre itération J. On dit parfois que s est "consommée" par J. C'est, en effet, une donnée de l'itération J. s est un "résultat intermédiaire". Nous dirons alors que s est une "séquence intermédiaire". Empruntons un exemple à J. BACKUS [Bac78]. L'expression FP

$(/+) \circ ((ALL \text{ mult}) \circ trans)$

appliquée à deux séquences $\langle x_1, x_2, \dots, x_n \rangle$ et $\langle y_1, y_2, \dots, y_n \rangle$ donne leur produit scalaire. L'expression suppose le calcul de deux séquences intermédiaires. L'une d'elles $\langle \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots, \langle x_n, y_n \rangle \rangle$ est le résultat de l'application de la primitive "trans". Cette séquence est une donnée de "ALL mult" qui a pour résultat $\langle x_1 y_1, x_2 y_2, \dots, x_n y_n \rangle$. Mais "/" "consomme" cette dernière pour donner le produit scalaire.

Ce point de vue facilite et clarifie l'expression des programmes mais a

première partie

l'inconvénient de multiplier les séquences intermédiaires. Si nous voulons un programme PASCAL [Jen74] calculant un produit scalaire de deux séquences données d'entiers, nous écrivons

```
PROGRAM SCAL;
VAR
  S,I,N,X,Y:INTEGER;
BEGIN
  READ(N);
  S:=0; FOR I:=1 TO N DO
    BEGIN
      READ(X,Y);
      S:=S+X*Y;
    END;
  WRITE(S);
END.
```

Le programme ne calcule qu'une séquence "intermédiaire" s . Les termes de s sont affectés à S . Si l'on nomme $x_1, x_2, \dots, x_n$ les données affectées à X et $y_1, y_2, \dots, y_n$ celles qui sont affectées à Y , s est la séquence

$$(s) \langle 0, x_1 y_1, x_1 y_1 + x_2 y_2, \dots, x_1 y_1 + x_2 y_2 + \dots + x_n y_n \rangle.$$

Notre objectif est de transformer une expression fonctionnelle d'un programme "avec des séquences", autrement dit d'un programme itératif. Nous considérons que le programme est exécuté par une machine du type "machine de VON NEUMANN" et est donc exécuté de manière séquentielle. Il devient donc souhaitable de minimiser le nombre des séquences intermédiaires résultats des itérations. Notre point de vue serait sans doute différent pour une exécution plus parallèle du programme. Dans une expression fonctionnelle, ces séquences sont les résultats des fonctions construites par des formes fonctionnelles telles que "ALL". Dans l'expression du produit scalaire

$$(e) \quad (/ +) \circ ((ALL \text{ mult}) \circ \text{trans}),$$

une séquence intermédiaire s_2

$$(s_2) \langle x_1 y_1, x_2 y_2, \dots, x_n y_n \rangle$$

est spécifiée par la fonction "ALL mult".

Notre objectif est de la supprimer. Il faut alors exprimer sans "ALL" la fonction composée f

$$(f) \quad (/ +) \circ (ALL \text{ mult})$$

qui a pour résultat la séquence s . Supposons que l'on possède une forme fonctionnelle E de construction de séquence nous permettant d'écrire la fonction (f) sous la forme $E(g)$ où g est une fonction à définir. Alors la fonction (e) s'écrit elle même sous la forme " $E(g) \circ \text{trans}$ ". Elle n'utilise pas la séquence s_2 . Nous transformons ainsi l'expression (e) en l'expression " $E(g) \circ \text{trans}$ ".

Le langage FP ne nous propose pas une forme fonctionnelle E qui convienne. Dans ce but nous enrichissons le langage FP avec une nouvelle forme fonctionnelle " $*$ ".

U.S. REDDY [Red82] propose une méthode de construction et de preuves de programmes qu'il appelle "programming with sequences" et nous lui avons emprunté le terme. Sa démarche est essentiellement semblable. Il utilise le langage FP et est conduit comme nous le faisons à l'enrichir avec des nouvelles formes fonctionnelles.

Dans le langage LUCID [Ash76], la programmation est aussi fondée sur les

suites. A l'origine, notre réflexion sur le mode de construction des programmes fut fortement influencée par LUCID. Mais les suites de LUCID sont toujours des suites infinies ce qui n'est pas conforme au résultat d'une itération qui se termine.

Nous avons depuis utilisé la programmation fonctionnelle qui nous permet d'avoir une algèbre simple des programmes qui facilite les preuves. De plus, l'introduction des formes fonctionnelles permet de donner une grande puissance au langage. Et enfin, le langage FP sert à exprimer les transformations voulues sur les programmes. Ces transformations auront pour but d'ôter les inefficacités introduites par le mode de construction du programme qui conduit à spécifier trop de séquences intermédiaires. Notre objectif particulier est donc de minimiser le nombre de ces séquences intermédiaires.

1.3. Programmation structurée

Nos idées sur la construction des programmes ont été aussi fortement influencées par les techniques de programmation structurée ([Dij76],[Hoa72],[Wir73],[Ars77]). La construction structurée peut être ascendante (bottom up) ou descendante (top down).

La construction structurée descendante d'un programme peut être faite en programmation fonctionnelle en introduisant un nom de fonction f dans une expression fonctionnelle e_1 . Cette fonction f sera définie ultérieurement par une expression fonctionnelle e_2 . Il suffit de pouvoir noter comme le fait LANDIN dans le langage ISWIN [Lan66]

$$e_1 \text{ where def } f = e_2$$

Une construction structurée ascendante s'établit très facilement par utilisation de la composition. On introduit la définition d'une fonction f que l'on compose ultérieurement avec une autre. Il suffit de noter

$$\text{let def } f = e_1$$

Pour construire une expression donnant le produit scalaire, on peut introduire f donnant la suite des produits des éléments

$$\text{let def } f = (\text{ALL mult}) \circ \text{trans}$$

puis on écrit $\text{def } e = (/+) \circ f$

Souvent, la construction structurée d'un programme utilise un mélange de des deux approches descendante et ascendante. Nous retrouvons ces idées dans les livres de W.H. BURGE [Bur75] et de P. HENDERSON [Hen80]. et dans les articles de J. BACKUS [Bac78], U.S. REDDY [Red82] et J. H. WILLIAMS [Wil80].

La programmation fonctionnelle conduit naturellement à une construction structurée des expressions des fonctions. Les formes fonctionnelles ont pour argument des fonctions. Naturellement, pour cette construction structurée, on est amené à donner un nom aux fonctions qui sont arguments des formes fonctionnelles. Le "découpage modulaire du programme" est donc une conséquence du choix des formes fonctionnelles du langage FP.

1.4. Contenu du chapitre

Pour coder les programmes itératifs, c'est-à-dire les programmes "consommant" et "produisant" des séquences, nous avons choisi un langage fonctionnel. Avec un tel langage, il est aisé d'avoir un petit ensemble de primitives sur les séquences et un petit ensemble de formes fonctionnelles qui permettent de coder les diverses formes des itérations. Le langage FP a été choisi en raison de son universalité. C'est aussi, à mon point de vue, un des langages fonctionnels les plus simples. En effet, le langage FP a un nombre fixé de formes fonctionnelles. Les formes fonctionnelles, qui sont des fonctions d'ordre 2, servent à coder les itérations et nous n'avons pas besoin de pouvoir définir d'autres fonctions d'ordre supérieur. Enfin, le formalisme du langage FP est simple et rigoureux, ce qui permet d'obtenir une algèbre simple des expressions fonctionnelles. Au paragraphe 2, nous commencerons par définir le langage FP que nous utiliserons. Nous devons en particulier définir l'ensemble des formes fonctionnelles qui codent les itérations.

Puis nous utiliserons ce langage et le mode de construction structurée "avec des séquences" dans quelques exemples au paragraphe 3. Ces exemples m'ont été suggérés par l'observation d'étudiants novices conduits à utiliser MEDEE ([Bel78],[Fin79],[Guy84],[Duc84]), qui est une méthode particulière de "construction structurée descendante avec des séquences", pour construire leurs programmes fondée sur les idées de C. PAIR [Pai79]. Il est "amusant" de comprendre le raisonnement suivi par les étudiants qui utilisent MEDEE.

Par exemple, pour calculer 2^n , on prend le dernier terme de la séquence $\langle 1, 2, 4, 8, 16, \dots, 2^n \rangle$.

Pour calculer $n!$, on prend le dernier terme de la séquence $\langle 1, 1, 2, 6, 24, \dots, n! \rangle$.

Pour calculer $2^n/n!$, il devient alors logique de diviser les deux derniers termes.

Du coup, on ne pense pas à la solution qui consiste à prendre le dernier terme de la séquence $s: \langle 1, 2, 4/2, 8/6, 16/24, \dots, 2^n/n! \rangle$. Dans cette solution, chaque terme $u_i = 2^i$ de s se calcule par la formule de récurrence $u_i = 2 * u_{i-1}$ avec "l'initialisation" $u_0 = 1$.

Les expressions des programmes ainsi obtenues sont correctes. Mais elles multiplient quelquefois maladroitement les séquences intermédiaires. Nous donnerons des expressions équivalentes qui minimisent le nombre de séquences intermédiaires. Ces exemples serviront à illustrer les résultats de notre travail dans les chapitres ultérieurs.

Grace aux formes fonctionnelles, le langage FP pourra aussi être utilisé pour exprimer les transformations à effectuer pour minimiser le nombre de séquences intermédiaires des expressions des programmes. Les équivalences entre les expressions fonctionnelles seront justifiées dans une interprétation algébrique. Désormais, nous considérerons toute expression fonctionnelle comme un terme d'une algèbre qui sera introduite au chapitre II.

2. Un langage pour exprimer et transformer les programmes itératifs

2.1. Les objets

Dans un langage FP, un objet peut être

-l'objet indéfini que l'on note "!"

-un atome

-un n-uplet d'objets

-une séquence d'objets.

L'ensemble des atomes contient toujours les booléens. Le booléen "FAUX" est noté "F". Le booléen "VRAI" est noté "T". Les atomes FP comprennent les entiers et les caractères. Les caractères seront notés entre guillemets. Nous pouvons ajouter les réels que nous noterons comme en PASCAL. Nous ajouterons en outre les chaînes de caractères notées elles aussi entre guillemets.

Le langage FP originel ne distingue pas les séquences de longueur arbitraire des n-uplets. Aussi il est possible de sélectionner le i-ème élément d'une séquence ou d'appliquer les opérations sur les séquences aux n-uplets. La distinction des deux types d'objets est nécessaire si l'on veut obtenir une théorie décidable pour ces deux types d'objets.

Les séquences sont notées entre "<" et ">". La séquence vide est notée "<>". Toute séquence contenant l'élément "!" vaut "!".

Les n-uplets sont notés entre "[" et "]". Tout n-uplet contenant l'élément "!" vaut "!". Les n-uplets ne sont pas plats; ils peuvent être au contraire de profondeur arbitraire. Par exemple, le triplet

première partie

[2,[3,5,[6,2]], [3,8]]

est de profondeur 3.

2.2. Les fonctions

Une fonction est définie en utilisant un ensemble de fonctions primitives et un petit ensemble de fonctions d'ordre supérieur qu'on appelle formes fonctionnelles. Une forme fonctionnelle a pour argument des fonctions et pour résultat une fonction. Une fonction est donc définie par application d'une forme fonctionnelle sur d'autres fonctions.

Dans les autres langages fonctionnels comme LISP [McC,62], on définit toujours une fonction par le résultat de son application. On dit alors que la fonction est définie par une définition applicative dite aussi définition "au niveau des objets" ("function-object definition" d'après J. BACKUS). Au contraire, une fonction qui est définie par application d'une forme fonctionnelle sur d'autres fonctions est dite définie "au niveau des fonctions" ("function level definition" d'après J. BACKUS).

Il est important de remarquer qu'une fonction FP n'admet qu'un unique objet en argument et n'a pour résultat qu'un seul objet. Il se peut cependant que l'objet résultat soit "!" (c'est à dire l'objet indéfini). En ce cas, si l'objet auquel la fonction s'applique est x, cela signifie que la fonction n'est pas définie pour l'objet x. Nous dirons alors que x n'est pas dans le domaine de définition de la fonction.

Appellons O l'ensemble des objets. Un sous ensemble D de O sera le domaine de définition d'une fonction f si et seulement si

première partie

-pour tout x élément de D , $f:x \Rightarrow y$ et y n'est pas "!",

-pour tout objet x de O non élément de D , $f:x \Rightarrow !$.

Rappelons que "!" est un élément de O . Nous pouvons donc considérer que f est de profil $O \rightarrow O$ même si son domaine est un sous ensemble strict de O : toute fonction FP est de profil $O \rightarrow O$. En conséquence il est toujours possible de composer deux fonctions FP.

Remarquons aussi qu'en FP, on ne peut définir de nouvelles fonctions d'ordre supérieur. Ceci n'est pas le cas pour les autres langages fonctionnels comme par exemple ML [Car80], HOPE [Bur80] ou SASL [Tur76]. Les formes fonctionnelles FP ne sont utilisables que pour construire des fonctions du premier ordre. Ainsi les expressions fonctionnelles pourront être vues comme des termes du premier ordre dans une algèbre libre.

Nous allons répertorier les primitives et les formes fonctionnelles que nous utiliserons. La plupart appartiennent au langage originel proposé par J. BACKUS. Nous justifierons l'introduction de nouvelles formes fonctionnelles. Si nous donnons la signification d'une fonction, nous en donnerons une définition applicative. Dans une définition applicative, " $\Rightarrow$ " signifiera "est défini par". Nous y noterons comme J. BACKUS l'application par ":". Donc " $f:x$ " remplacera la notation habituelle " $f(x)$ ".

3. Les primitives de base

Nous utiliserons

a) l'identité notée "id" telle que, pour tout objet x

première partie

id: $x \Rightarrow x$

b) des primitives sur les entiers ou les réels comme "+", "-", "mult", "div" (division entière), "mod" (reste de la division entière) et "/" (division entre réels).

On a par exemple

+ : [3,5] $\Rightarrow$ 8

- : [3,5] $\Rightarrow$ -2

mult : [5,3] $\Rightarrow$ 15

div : [11,3] $\Rightarrow$ 3

mod : [11,3] $\Rightarrow$ 2

/ : [5,2] $\Rightarrow$ 2.5

c) des primitives sur les booléens comme "and", "or" et "not", avec leur sens habituel,

d) des primitives de comparaison comme "eq0" (égal à 0), "eq" (égal), "gt" (plus grand que), "ge" (plus grand ou égal), "lt" (plus petit que) et "le" (plus petit ou égal).

Une définition rigoureuse de ces primitives ne nous semble pas nécessaire, mais nous devons remarquer que cette définition respecte les contraintes de toutes les fonctions FP : elles sont de profil $O \rightarrow O$ même si leur domaine de définition est strictement inclus dans O . Nous aurons par exemple

+ : [3,4,5] $\Rightarrow$!

+ : [7,"maison"] $\Rightarrow$!

première partie

$+ : [7,!] == + : ! == !$

Ce dernier exemple montre que "+" est une fonction stricte. Il en est de même de toute fonction FP. Nous expliquerons pourquoi au paragraphe 4.

d) les sélecteurs, notés "?1", "?2", "?3",...

Si $x_1, x_2, x_3, \dots, x_n$ sont différents de "!", nous avons

?1 : $[x_1, x_2, x_3, \dots, x_n] == x_1$

?2 : $[x_1, x_2, x_3, \dots, x_n] == x_2$

?3 : $[x_1, x_2, x_3, \dots, x_n] == x_3, \dots$

et, dans les autres cas, le résultat est "!".

Les sélecteurs sont indispensables pour écrire une expression fonctionnelle. Ils permettent de noter les éléments d'un n-uplet argument de la fonction. Si une fonction f a pour domaine de définition des n-uplets, nous dirons qu'elle est d'arité n . Nous avons par exemple une forme fonctionnelle notée "*" qui s'applique à une fonction d'arité deux. Si l'on écrit "*f", f a donc pour argument un couple. Quand on exprime f , les deux éléments du couple seront les résultats des sélecteurs "?1" pour le premier et "?2" pour le deuxième. Considérons l'expression

$*(+ o [?1, \text{mult } o ?2]),$

que l'on peut écrire

$*f \text{ where } \text{def } f = + o [?1, \text{mult } o ?2].$

Le sélecteur "?1" joue le rôle du premier élément du couple en argument et le sélecteur "?2" celui du deuxième élément de ce couple dans l'expression

première partie

de f .

3.1. Les primitives sur les séquences

Leur choix conditionne l'écriture des expressions fonctionnelles qui correspondent aux programmes itératifs. Nous utiliserons

a) le premier terme d'une séquence, noté hd ,

Si $x_1, x_2, \dots, x_n$ sont différents de "!" et n supérieur ou égal à 1, nous avons

$hd : \langle x_1, x_2, \dots, x_n \rangle == x_1$

dans tous les autres cas le résultat est "!".

b) le dernier terme d'une séquence, noté "last",

Si $x_1, x_2, \dots, x_n$ sont différents de "!" et n supérieur ou égal à 1, nous avons

$\text{last} : \langle x_1, x_2, \dots, x_n \rangle == x_n$

dans tous les autres cas le résultat est "!".

"last" est souvent employé pour obtenir le dernier terme d'une séquence résultat d'une itération.

c) la primitive qui ajoute un terme devant une séquence est notée "cons",

Si z est différent de "!", nous avons

$\text{cons} : [z, \langle \rangle] == \langle z \rangle$

première partie

Si $z, x_1, x_2, \dots, x_n$ sont différents de "!", nous avons

$$\text{cons} : [z, \langle x_1, x_2, \dots, x_n \rangle] == \langle z, x_1, x_2, \dots, x_n \rangle$$

dans tous les autres cas le résultat est "!".

"cons" est notée "append!" dans le langage FP original. "cons" sera utilisée pour définir les nouvelles formes fonctionnelles "*" et "WH".

d) le reste d'une séquence, noté "tl",

Si $x_1, x_2, \dots, x_n$ sont différents de "!", nous avons

$$\text{tl} : \langle x_1 \rangle == \langle \rangle$$

$$\text{tl} : \langle x_1, x_2, x_3, \dots, x_n \rangle == \langle x_2, x_3, \dots, x_n \rangle$$

dans tous les autres cas, le résultat est "!".

L'itération "*f" place comme premier terme de la séquence résultat "l'initialisation de l'itération". "tl" sera utilisée pour l'éliminer.

e) la primitive, notée "trans", est d'arité multiple,

Quand elle s'applique à deux séquences de même longueur, "trans" donne la séquence des couples de leurs termes

$$\text{trans} : [\langle \rangle, \langle \rangle] == \langle \rangle$$

$$\text{trans} : [\langle x \rangle, \langle y \rangle] == \langle [x, y] \rangle$$

$$\text{trans} : [\langle x_1, x_2, \dots, x_n \rangle, \langle y_1, y_2, \dots, y_n \rangle] == \langle [x_1, y_1], [x_2, y_2], \dots, [x_n, y_n] \rangle,$$

quand elle s'applique à trois séquences de même longueur, elle donne la

première partie

séquence des triplets de leurs termes

$$\text{trans} : [\langle \rangle, \langle \rangle, \langle \rangle] == \langle \rangle$$

$$\text{trans} : [\langle x \rangle, \langle y \rangle, \langle z \rangle] == \langle [x, y, z] \rangle$$

$$\text{trans} : [\langle x_1, x_2, \dots, x_n \rangle, \langle y_1, y_2, \dots, y_n \rangle, \langle z_1, z_2, \dots, z_n \rangle]$$

$$== \langle [x_1, y_1, z_1], [x_2, y_2, z_2], \dots, [x_n, y_n, z_n] \rangle$$

quand elle s'applique à m séquences de même longueur, elle a pour résultat la séquence des m-uplets de leurs termes

$$\text{trans} : [\langle \rangle, \langle \rangle, \dots, \langle \rangle] == \langle \rangle$$

$$\text{trans} : [\langle x_1 \rangle, \langle x_2 \rangle, \dots, \langle x_m \rangle] == \langle [x_1, x_2, \dots, x_m] \rangle$$

$$\text{trans} : [\langle x_{11}, x_{12}, \dots, x_{1n} \rangle, \langle x_{21}, x_{22}, \dots, x_{2n} \rangle, \dots, \langle x_{m1}, x_{m2}, \dots, x_{mn} \rangle]$$

$$== \langle [x_{11}, x_{21}, \dots, x_{m1}], [x_{12}, x_{22}, \dots, x_{m2}], \dots, [x_{1n}, x_{2n}, \dots, x_{mn}] \rangle.$$

C'est alors la transposition de la matrice des x_{ij} .

Dans les autres cas, elle a pour résultat "!".

Nous avons déjà cité l'expression "/* o ((ALL mult) o trans)" qui, appliquée à deux séquences d'entiers, a pour résultat leur produit scalaire. "trans" permet d'obtenir la séquence des couples des deux séquences données. Cette séquence est alors argument de "ALL mult".

f) la primitive, notée "distl",

permet de coupler un même objet à tous les termes d'une séquence.

première partie

Si z n'est pas "!", nous avons

$\text{dist1} : [z, \langle \rangle] == \langle \rangle.$

Si $z, x_1, x_2, \dots, x_n$ ne sont pas "!", nous avons

$\text{dist1} : [z, \langle x_1, x_2, \dots, x_n \rangle] == \langle [z, x_1], [z, x_2], \dots, [z, x_n] \rangle$

et autrement, le résultat est "!".

g) la primitive qui juxtapose deux séquences notée "append"

$\text{append} : [\langle \rangle, \langle \rangle] == \langle \rangle$

Si $x_1, x_2, \dots, x_n$ ne sont pas "!", nous avons

$\text{append} : [\langle x_1, x_2, \dots, x_n \rangle, \langle \rangle] == \langle x_1, x_2, \dots, x_n \rangle$

$\text{append} : [\langle \rangle, \langle x_1, x_2, \dots, x_n \rangle] == \langle x_1, x_2, \dots, x_n \rangle$

Si de plus $y_1, y_2, \dots, y_m$ ne sont pas "!", nous avons

$\text{append} : [\langle x_1, x_2, \dots, x_n \rangle, \langle y_1, y_2, \dots, y_m \rangle] ==$

$\langle x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m \rangle$

et autrement le résultat est "!".

C'est le "append" de LISP.

h) la primitive "iota",

"iota", appliquée à un entier positif n , fournit la séquence des n entiers consécutifs de 1 à n . On trouve aussi cette primitive "iota" en APL [Gui78].

première partie

$\text{iota} : n == \langle 1, 2, 3, \dots, n \rangle$

autrement, son résultat est "!".

3.2. Les formes fonctionnelles de base

Elles s'appliquent à des fonctions FP et ont pour résultat des fonctions FP.

Nous trouvons

a) la composition de deux fonctions notée "_o_",

$(f \circ g) : x == f : (g : x)$

Dans une expression fonctionnelle, "o" sera toujours considérée comme l'opération la moins prioritaire ce qui évitera l'écriture de quelques parenthèses.

REMARQUE

Pour indiquer l'écriture préfixée, infixée ou autre d'une forme fonctionnelle, nous utilisons la notation de OBJ [Gog79]. Le symbole "_" permet d'indiquer la place d'un argument. "_o_" signifie que la forme fonctionnelle de symbole "o" est d'arité deux avec une notation infixée.

b) la construction d'arité multiple notée "[_..._]",

Si x n'est pas "!", nous avons

$[f1, f2, \dots, fn] : x == [f1:x, f2:x, \dots, fn:x]$

première partie

et dans les autres cas, le résultat est "!".

Cette forme fonctionnelle permet de "construire" un n-uplet argument d'une fonction. Par exemple nous écrivons l'expression

$$+ o [?1, \text{mult } o [?2, ?2]]$$

pour une fonction qui appliquée au couple $[x, y]$ a pour résultat $x + y^2$

Appliquons la à $[2, 5]$.

$$+ o [?1, \text{mult } o [?2, ?2]] : [2, 5] == + : [?1:[2, 5], \text{mult} : [?2:[2, 5], ?2:[2, 5]]]$$

$$== + : [2 , \text{mult} : [5 , 5]]$$

$$== + : [2, 25]$$

$$== 27$$

c) la forme fonctionnelle notée " $\bar{_}$ ",

Elle permet d'obtenir la fonction constante associée à un objet.

Pour tout objet y différent de "!", on a

$$x \bar{_} : y == x$$

$$\text{et } x \bar{_} : ! == !$$

Par exemple, " $+ o [1 \bar{_}, i]$ " ajoute 1 à l'entier résultat d'une fonction i .

d) la forme conditionnelle notée " $\bar{_} \rightarrow \bar{_}; \bar{_}$ " est définie par

Si $p : x == T$ alors $(p \rightarrow f; g) : x == f : x$,

si $p : x == F$ alors $(p \rightarrow f; g) : x == g : x$,

et autrement le résultat est "!".

première partie

L'expression " $\text{gt} \rightarrow ?1 ; ?2$ " appliquée à deux entiers donne leur maximum.

3.3. Les formes fonctionnelles construisant des fonctions qui s'appliquent à des séquences

La seule que nous emprunterons au langage FP original est "ALL".

a) "apply to all" notée "ALL" est "mapcar" en LISP

Si f est une fonction FP, nous avons

$$(ALL f) : \langle \rangle == \langle \rangle$$

$$(ALL f) : \langle x_1, x_2, \dots, x_n \rangle == \langle f : x_1, f : x_2, \dots, f : x_n \rangle$$

et autrement "ALL f " a pour résultat "!".

L'expression " $ALL \bar{2} \circ \text{iota}$ " est une fonction qui, appliquée à un entier positif n , rend une séquence de 2, " $\langle 2, 2, \dots, 2 \rangle$ ", de longueur n .

Appliquons la à 5

$$(ALL \bar{2} \circ \text{iota}) : 5 == ALL \bar{2} : (\text{iota} : 5)$$

$$== ALL \bar{2} : \langle 1, 2, 3, 4, 5 \rangle$$

$$== \langle \bar{2} : 1, \bar{2} : 2, \bar{2} : 3, \bar{2} : 4, \bar{2} : 5 \rangle$$

$$== \langle 2, 2, 2, 2, 2 \rangle$$

L'expression " $ALL ?1 \circ (\text{dist1 } o [\bar{2}, \text{iota}])$ " est la même fonction.

Appliquons la aussi à 5

première partie

```
ALL?1 o (dist1 o [2-,iota]):5
```

```
== ALL?1:(dist1:[2-:5,iota:5])
== ALL?1:(dist1:[2-:5,<1,2,3,4,5>])
== ALL?1:(dist1:[2,<1,2,3,4,5>])
== ALL?1:<[2,1],[2,2],[2,3],[2,4],[2,5]>
== <?1:[2,1],?1:[2,2],?1:[2,3],?1:[2,4],?1:[2,5]>
== <2      ,2      ,2      ,2      ,2>
```

Nous n'utiliserons pas les formes fonctionnelles FP "insert1" notée aussi "/", "insertr" notée aussi "\" et "while" mais nous allons les remplacer par deux autres formes fonctionnelles "*" et "WH".

b) "L'itéré" noté "*_"

Débutons par un exemple. Considérons l'itération PASCAL

```
(P)
S:= 0;
FOR I:= 1 TO 5 DO
  BEGIN
    READ(X);
    S:=S+X;
  END;
```

"L'itéré" peut s'appliquer à la fonction primitive "+". La fonction "*" doit correspondre à cette itération. Nous considérons que le résultat de l'itération est une séquence s qui est affectée à S. Si l'entrée est la séquence donnée <2,3,7,4,6>, la séquence s est

```
<0,   valeur initiale de l'itération
2,   résultat du calcul de 0 + 2
5,   résultat du calcul de 2 + 3
12,  résultat du calcul de 5 + 7
16,  résultat du calcul de 12 + 4
22>  résultat du calcul de 16 + 6
```

La fonction "*", appliquée à la séquence <2,3,7,4,6>, a aussi pour

première partie

résultat s

```
** : [0,<2,3,7,4,6>] == <0,2,5,12,16,22>.
```

Remarquons que la séquence donnée peut aussi s'écrire

```
cons : [2.<3,7,4,6>].
```

Nous pourrions définir "*" par une définition récursive sur la structure de cette séquence donnée.

Définition de "*"f

a) Dans le cas où la séquence donnée est vide, la séquence résultat n'a qu'un élément. Cet élément correspond à "la valeur initiale" pour une itération classique.

```
*f : [z,<>] == <z>
```

b) Dans le cas où la séquence donnée n'est pas vide, elle peut s'écrire "cons : [x,a]". On peut alors donner la définition récursive suivante de "*"f

```
*f : [z,cons:[x,a]] == cons : [z, *f:[f:[z,x],a]]
```

c) Dans les autres cas, le résultat est "!".

Nous substituons "*" à la forme "insert1" notée "/" en FP. Nous allons tout d'abord étudier "/". Nous verrons alors pourquoi nous avons introduit un "itéré" différent.

Notons déjà que "/"f associe ses opérandes de gauche à droite

première partie

$$/f : \langle x_1, x_2, x_3, x_4, x_5 \rangle == f:[f:[f:[f:[x_1, x_2], x_3], x_4], x_5]$$

f à pour arité 2. L'expression $"/+ "$ est l'opérateur $"\Sigma "$ des mathématiques. Un exemple peut être

$$/+ : \langle 2, 3, 7, 4, 6 \rangle == 22.$$

Le calcul est fait de gauche à droite : $((((2+3)+7)+4)+6)$.

En APL, on trouve un opérateur $"/+ "$, qui donne le même résultat mais le calcul est fait de droite à gauche : $(2+(3+(7+(4+6))))$. Il en est de même avec "reduce" dans le livre de P. HENDERSON [Hen80].

En programmation classique, $"/+ "$ correspond aussi à l'itération Pascal (ρ) citée ci-dessus.

Contrairement à $"*f "$, la fonction $"/+ "$ n'a pas pour résultat la séquence intermédiaire $s : \langle 0, 2, 5, 12, 16, 22 \rangle$. $"/+ "$ donne directement pour résultat 22 qui est le dernier terme de s.

$"*f "$, comme $"/f "$, associe ses opérandes de gauche à droite et la fonction argument f est d'arité 2

Mais $"/f "$ ne nous satisfait pas totalement.

Nous avons cité l'exemple du produit scalaire avec une expression utilisant $"/+ "$.

$$/+ o (ALL mult o trans)$$

Si nous voulons éliminer la séquence intermédiaire spécifiée par "ALL mult", la même fonction peut être écrite uniquement avec $"/+ "$, elle s'écrit

première partie

$$/(+ o [?1, mult o ?2]) o (cons o [0^-, trans])$$

Cette expression utilise la primitive "cons". Elle fait apparaître l'initialisation de la somme à 0.

Nous voulons mettre en évidence la valeur initiale sans utiliser "cons"

Notons que U.S. REDDY [Jay83] considère une forme "insertl" modifiée qui lui permet de mettre en évidence l'initialisation

$$insertl f : [z, \langle x_1, x_2, x_3, x_4 \rangle] == f:[f:[f:[f:[z, x_1], x_2], x_3], x_4]$$

L'argument z donne la valeur initiale.

Si nous voulons écrire une fonction donnant le produit scalaire de deux séquences nous avons alors le choix entre deux formules. "ALL mult" a pour résultat une séquence qui peut être le deuxième argument de "insertl +". Ainsi l'itération "insertl +" est capable de "consommer" une séquence résultat de "ALL mult". Nous écrivons alors une première formule qui utilise la composition possible entre "insertl +" et "ALL mult"

$$insertl + o [0^-, ALL mult o trans]$$

Une deuxième formule peut être écrite sans "ALL mult"

$$insertl (+ o [?1, mult o ?2]) o [0^-, trans]$$

Ces formules font toutes deux apparaître l'initialisation à 0. La première formule spécifie la séquence intermédiaire des produits $\langle x_1 y_1, x_2 y_2, \dots, x_n y_n \rangle$ résultat de "ALL mult o trans", ce que ne fait pas la deuxième.

première partie

"*f" fait apparaître "l'initialisation".

REMARQUE

Il pourrait sembler plus simple de faire apparaître "l'initialisation" comme deuxième argument de "*" plutôt que comme argument de "*f". On écrirait par exemple " $*(+,0^-)$ ". Mais alors l'objet initial ne peut être que le résultat d'une fonction appliquée à des constantes ce qui est fréquent mais n'est pas général. Une fonction qui calcule le minimum d'une séquence d'entiers dont on n'a pas d'ordre de grandeur s'écrit

$\text{last } o \ (*f \ o \ [hd,tl]) \ \text{where} \ \text{def } f = lt \rightarrow ?1;?2$

L'expression ne peut pas s'écrire " $*(f,hd) \ o \ tl$ ".

Le plus important est que nous voulons pouvoir composer les diverses sortes d'itérations entre elles.

La première formule de l'exemple ci-dessus compose une fonction définie par "ALL mult" (c'est à dire une itération) avec la fonction définie par "insertl +" (c'est à dire une itération). L'itération "insertl +" "consomme" la séquence des produits résultat de "ALL mult". Nous pouvons composer "insertl g" avec "ALL f" car "ALL f" rend une séquence. Mais nous ne pouvons pas composer ainsi "ALL g" avec "insertl f" ou "insertl g" avec "insertl f" puisque "insertl f" n'a pas pour résultat une séquence.

Or nous considérons qu'une itération calcule toujours une séquence. Par exemple, l'itération PASCAL (P) déjà citée dans ce paragraphe

première partie

```
S:=0;
FOR I:=1 TO 5 DO
  BEGIN
    READ(X);
    S:=S+X
  END;
```

calcule la séquence s

$\langle 0,2,5,12,16,22 \rangle$

qui est affectée à S quand la séquence d'entrée est $\langle 2,3,7,4,6 \rangle$.

Il est naturel qu'une fonction qui exprime une itération ait pour résultat une séquence. Une conséquence est que l'on peut toujours composer entre elles les fonctions qui expriment les itérations. Aussi nous décidons que la fonction "*f" a pour résultat une séquence. Nous avons donc par exemple

$*f : [z, \langle x_1, x_2, x_3 \rangle] == \langle z, f : [z, x_1], f : [f : [z, x_1], x_2], f : [f : [f : [z, x_1], x_2], x_3] \rangle$.

L'argument z donne la valeur initiale et le résultat de "*f" est une séquence. Ceci peut sembler inutile dans un langage fonctionnel mais est indispensable pour étudier la construction des programmes itératifs. U.S. REDDY [Jay83] note "cuml" notre itéré "*".

Quand on désire uniquement le dernier terme d'une séquence, on doit utiliser la primitive "last".

$(\text{last } o \ **) : [0, \langle 2,3,7,4,6 \rangle] == 22$

Quand on ne veut pas la valeur initiale dans la séquence résultat, on utilise la primitive "tl".

première partie

$(t1 \text{ o } *) : [0, \langle 2, 3, 7, 4, 6 \rangle] == \langle 2, 5, 12, 16, 22 \rangle$

Le produit scalaire peut s'écrire à l'aide d'une formule utilisant "ALL mult"

$\text{last o } (* \text{ o } [0^-, \text{ALL mult o trans}])$

ou à l'aide d'une formule sans "ALL mult"

$\text{last o } (*(\text{ o } [?1, \text{mult o } ?2]) \text{ o } [0^-, \text{trans}])$

Dans les deux cas, on utilise "last".

c) "while" noté "WH__"

Comme " $*f$ ", " $\text{WH}(p, f)$ " associe les opérandes de f de gauche à droite, fait apparaître l'initialisation et engendre une fonction ayant pour résultat une séquence. Mais il a un paramètre supplémentaire p qui est un prédicat qui permet l'arrêt. La fonction " $\text{WH}(p, f)$ " a les mêmes arguments que l'itéré, une valeur initiale et une séquence. La fonction f et le prédicat p sont d'arité 2 et s'appliquent aux mêmes arguments.

Avec " $*_$ ", nous avons par exemple

$*+ : [0, \langle 2, 3, 7, 4, 6 \rangle] ==$

$\langle 0,$	valeur initiale	
	2,	$0 + 2$
	5,	$2 + 3$
	12	$5 + 7$
	16	$12 + 4$
	22>	$16 + 6$

et avec "WH__", nous avons plutôt

première partie

$\text{WH}(p, +) : [0, \langle 2, 3, 7, 0, 4, 6 \rangle]$ where def $p = \text{not o } (\text{eq0 o } ?2)$

séquence résultat résultat du prédicat

$== \langle 0,$	valeur initiale	
2,	$0 + 2$	T (2 n'est pas 0)
5,	$2 + 3$	T (3 n'est pas 0)
12	$5 + 7$	T (7 n'est pas 0)
>		F (0 est égal à 0)

Cela correspond à l'itération PASCAL

```
S := 0; READ(X);
WHILE X # 0 DO
  BEGIN
    S:=S+X;
    READ(X)
  END;
```

qui, avec les données "2,3,7,0", affecte la séquence $\langle 0, 2, 5, 12 \rangle$ à S.

Dans ce premier exemple le prédicat p s'applique aux éléments de la séquence argument. Mais il peut aussi s'appliquer aux éléments de la séquence des termes calculés comme dans l'exemple

$\text{WH}(p, \text{mult}) \text{ o } [1^-, \text{iota}]$ where def $p = \text{lt o } [?1, 1000^-]$.

Appliquée à un entier n , cette fonction donne la séquence $\langle 1!, 2!, \dots, n! \rangle$ quand $n!$ est inférieur à 1000 (c'est à dire pour n inférieur à 6) et $\langle 1!, 2!, 3!, 4!, 5!, 6! \rangle$ quand n est supérieur ou égal à 6.

Dans le cas général, le prédicat peut s'appliquer au terme calculé et à l'élément de la séquence argument de "WH__".

REMARQUE

première partie

Les expressions qui utilisent "WH" codent des programmes itératifs qui se terminent: si le prédicat n'est pas vérifié, l'itération s'arrête à la fin de la séquence en argument.

Définition de "while"

$WH(p,f) : [z, \langle \rangle] == \langle z \rangle,$

si $p:[z,x] == T$ alors

$WH(p,f) : [z, cons:[x,a]] == cons:[z, WH(p,f) : [f:[z,x], a]],$

si $p:[z,x] == F$ alors $WH(p,f) : [z, cons:[x,a]] == \langle z \rangle$

et, dans les autres cas, le résultat est "!".

d) Discussion

Nous discutons sur le choix des formes fonctionnelles permettant de construire les fonctions sur les séquences correspondant aux itérations

Nous n'avons pas trouvé de formes fonctionnelles semblables à notre "WH" chez d'autres auteurs. Cependant, nous trouvons des formes fonctionnelles très variées pouvant servir de "tant que". Nous allons examiner deux exemples empruntés à U. S. REDDY [Red82].

- Une forme fonctionnelle, notée "first" par U. S. REDDY

"first" a pour argument un prédicat. La fonction "first p" a pour résultat le premier (de gauche à droite) élément de la séquence argument qui vérifie le prédicat p s'il existe, sinon son résultat est "!". Par exemple avec un prédicat p qui teste si un entier est pair, nous avons

$first\ p : \langle 3,7,5,4,3,8,9 \rangle == 4$

On peut définir p par

$where\ def\ p = eq0\ o\ (mod\ o\ [id, 2])$

On comprend l'aisance que peut donner l'utilisation de "first" dans de nombreux problèmes (recherche dans une liste du premier élément de cette liste qui vérifie un prédicat, recherche associative d'un élément d'une table,...). Une forme similaire existe en LUCID.

Avec "WH", il faut écrire la formule suivante, qui utilise le même prédicat p, pour obtenir les mêmes résultats

$(p\ o\ f) \rightarrow f ; !$

$where\ def\ f = last\ o\ (WH(not\ o\ (p\ o\ ?1), ?2)\ o\ [hd, tl]).$

En tenant compte de la définition de "WH", la fonction f a pour résultat le premier élément de la séquence donnée vérifiant p, s'il existe. Sinon f a pour résultat le dernier élément de cette séquence. L'expression " $(p\ o\ f) \rightarrow f ; !$ " est une autre expression de la fonction "first p".

L'on pourrait ainsi fournir des formes fonctionnelles agréables à utiliser pour construire des fonctions correspondantes à chaque type de problème. Mais alors, on augmenterait sensiblement le nombre des formes fonctionnelles du langage. Nous n'avons pas voulu le faire car notre but n'était pas de décrire un tel langage. Le langage FP n'est pas utilisé comme langage de construction de programmes. Il est utilisé uniquement comme formalisme pour exprimer les programmes et pour avoir une base de travail suffisante pour étudier les transformations qui minimisent les séquences intermédiaires introduites dans les programmes itératifs.

On peut d'ailleurs remarquer que l'utilisation de "first" est délicate

car il est facile d'oublier que la fonction "first p" a pour résultat "!" dans le cas où aucun terme de la séquence en argument ne vérifie p. On peut aussi remarquer que le même problème se pose pour utiliser "->_;" quand on n'a pas l'habitude d'une conditionnelle qui peut avoir pour résultat "!". Il est assez difficile de savoir exactement quelles sont les bonnes formes fonctionnelles à introduire.

Les autres auteurs ne se préoccupent pas uniquement de la construction de programmes. Quand U.S. REDDY étudie les transformations des formules récursives en formules itératives [Jay83], il introduit une forme fonctionnelle notée "seq" qui engendre une séquence. "seq" est adaptée à l'expression de ces transformations.

● Utilisons "seq" dans un exemple simple

Disons brièvement que "seq(f,p)" s'applique à un objet et engendre une séquence d'éléments résultats successifs de la fonction f et ceci à partir de cet objet initial. Le prédicat p arrête la production. La fonction

seq (f,eq0) where def f = - o [id,1~]

appliquée à un entier n a pour résultat la séquence des n entiers consécutifs en décroissant à partir de n.

seq (f,p) : 5 == <5,4,3,2,1>

Nous écrivons la même fonction avec l'expression

*(- o [?1,1~]) o [id,iota]

Nous avons choisi "ALL", "*" et "WH" dans le but de pouvoir considérer

les expressions les plus courantes des programmes itératifs construits par une méthode structurée avec des séquences.

Nous avons aussi voulu que "ALL", "*" et "WH" nous permettent d'avoir une algèbre de programmes suffisamment riche pour nous fournir une bonne base pour l'étude des transformations des programmes itératifs. Ces transformations ont pour but de minimiser les séquences intermédiaires spécifiées par l'expression. Nous avons donc veillé à proposer le plus de liberté possible pour composer entre elles les itérations.

Et enfin, nous n'avons pas voulu grossir le nombre des formes fonctionnelles utilisées dans ce travail. En effet, certaines idées de formes fonctionnelles pourtant intéressantes ont été abandonnées avec regret.

4. Construction d'expressions fonctionnelles simples

Nous allons étudier la construction de quelques expressions fonctionnelles simples qui nous serviront d'exemples au cours de notre travail. Il n'est pas indispensable de lire ce paragraphe pour comprendre la suite du travail. Cependant il nous permet de mieux comprendre son objectif. Au départ, l'idée est venue de problèmes pratiques de pédagogie rencontrés au cours de l'enseignement de la programmation avec une méthode structurée descendante. Nous terminons cette série d'exemples par celui qui nous a donné l'idée de ce travail [Bel80] au paragraphe 3.6. Les exemples d'expressions reflètent les inefficacités qui peuvent être introduites dans les programmes itératifs avec un mode de raisonnement structuré à partir des séquences. Les expressions obtenues spécifient trop de séquences intermédiaires.

Pour une expression e , nous compterons les séquences résultats des fonctions issues des formes fonctionnelles "ALL", "*" et "WH" qui codent les itérations. Pour cela, il ne suffit pas de compter les occurrences de "ALL", "*" et "WH" car nous entendons compter les séquences résultats de chaque exécution d'une itération pour une interprétation de l'expression par une machine de type VON NEUMANN, ceci pour rendre compte de ce qui se passe pour les programmes. Pour les exemples cités, qui ne comportent pas, pour la plupart, de conditionnelles, nous pouvons obtenir exactement le nombre de séquences intermédiaires issues des itérations. Pour une expression e , nous noterons ce nombre $I(e)$.

Nous compterons de la même manière le nombre des séquences intermédiaires qui sont les résultats des itérations mais aussi des expressions des primitives "trans", "distl", "tl", "iota", "append" et "cons". Pour une expression e , nous noterons $N(e)$ ce nombre.

Nous donnerons pour chaque exemple plusieurs expressions. Chaque expression correspond à un raisonnement différent. Nous avons effectivement constaté que ces divers raisonnements sont faits par nos étudiants novices. Nous ne prétendons pas en faire un catalogue exhaustif. Nos transformations devront donner une expression e qui minimise $I(e)$ et $N(e)$.

Conseil pour le lecteur

Ce qui rend souvent pénible la lecture des expressions FP est l'utilisation des sélecteurs. Ceux-ci servent à noter les éléments d'un n -uplet argument d'une fonction. Quand on écrit l'expression d'une fonction f argument de "*", il faut se souvenir du schéma suivant:

*f : [z, <a1, a2, a3, ..., an>] == <r0, r1, r2, ..., rn>

indice	argument de f		résultat
i	[_,_]	?1 ?2	ri
0			z
1	[z, a1]	z a1	r1=f:[z, a1]
2	[r1, a2]	r1 a2	r2=f:[r1, a2]
3	[r2, a3]	r2 a3	r3=f:[r2, a3]
...			
n	[rn-1, an]	rn-1 an	rn=f:[rn-1, an]

4.1. Les séquences intermédiaires sont issues des compositions d'itérations

Nous proposons deux expressions pour calculer 2^n et deux expressions pour calculer x^n .

L'une compose une itération ayant pour résultat une séquence de 2 (pour le premier exemple) ou de x (pour le deuxième exemple) avec une itération qui les multiplie.

L'autre propose une seule itération ayant pour résultat la séquence des puissances successives de 2 (pour le premier exemple) ou des puissances successives de x (pour le deuxième exemple). Cette deuxième expression est optimum pour le nombre de séquences intermédiaires.

a) Exemple 1

Pour trouver une expression fonctionnelle calculant 2^n , on introduit une fonction n qui a pour résultat l'exposant n car ce sera plus lisible dans l'expression que "id" ou "?1".

● ON PEUT RAISONNER SELON UN MODE DE CONSTRUCTION ASCENDANT

On sait que "iota o n" donne la séquence $\langle 1, 2, 3, \dots, n \rangle$.

On peut alors obtenir une séquence $\langle 2, 2, 2, \dots, 2 \rangle$, ayant pour longueur n, comme résultat de la fonction composée de "iota o n" avec "ALL 2"

let def f = ALL 2⁻ o (iota o n)

Il suffit alors de multiplier

*mult o [1⁻, f] donne $\langle 1, 2, 4, 8, \dots, 2^n \rangle$

2ⁿ est le dernier terme de cette séquence

last o (*mult o [1⁻, f]) donne 2ⁿ

L'EXPRESSION TROUVEE EST

(1) last o (*mult o [1⁻, ALL 2⁻ o (iota o n)])

Considérons l'interprétation de cette expression sur un objet z pour lequel elle n'a pas pour résultat "!" (autrement dit pour lequel elle est définie)

n	iota	ALL 2 ⁻	
z	---> n	---> $\langle 1, 2, \dots, n \rangle$	---> $\langle 2, 2, \dots, 2 \rangle$
			*mult
			---> $\langle 1, 2, 4, \dots, 2^n \rangle$
			last
			---> 2 ⁿ

Nous comptons donc deux séquences intermédiaires résultats de "ALL 2⁻" et de "*mult"

première partie

N(1) = I(1) = 2.

● ON PEUT RAISONNER AVEC UN MODE DE CONSTRUCTION DESCENDANT

Si on définit une fonction f ayant pour résultat

$\langle 1, 2, 4, 8, \dots, 2^n \rangle$,

l'expression "last o f" donnera 2ⁿ. f peut s'exprimer par une itération "*g". Pour i de 1 à n, g a un premier argument 2ⁱ⁻¹ et a pour résultat 2ⁱ. La valeur initiale est 2⁰. On peut écrire

where def f = *g o [1⁻, iota o n]

g doit multiplier par 2 son premier argument. g s'écrit donc

where def g = mult o [?, 2⁻]

L'EXPRESSION TROUVEE EST ALORS

(2) last o (*(mult o [?, 2⁻]) o [1⁻, iota o n])

Le schéma du calcul est le suivant

n	iota	
z	---> n	---> $\langle 1, 2, \dots, n \rangle$
		*(mult o [?, 2 ⁻])
	1 ⁻	---> $\langle 1, 2, 4, \dots, 2^n \rangle$
		last
		---> 2 ⁿ

L'expression (2) ne spécifie qu'une séquence intermédiaire. La séquence de 2 qui est introduite par "ALL 2⁻" dans l'expression (1) a disparu:

N(2) = I(2) = 1

première partie

b) Exemple 2

Pour une expression calculant x^n , une difficulté nouvelle apparaît avec l'argument supplémentaire x .

On introduit deux fonctions x et n ayant respectivement pour résultat x et n . Ces résultats sont obtenus pour toute donnée d'un objet z du domaine de définition de l'expression de calcul de x^n . x et n sont plus agréables à retrouver dans cette expression que "?1" et "?2".

● EN CHERCHANT SELON UN MODE DE CONSTRUCTION ASCENDANT

La primitive "distl" peut nous permettre d'obtenir une séquence de couples $[x, i]$

$\text{distl } o [x, \text{iota } o n]$ donne $\langle [x, 1], [x, 2], [x, 3], \dots, [x, n] \rangle$

On peut donc écrire une fonction f ayant pour résultat une séquence de x ayant pour longueur n .

let def $f = \text{ALL ?1 } o (\text{distl } o [x, \text{iota } o n])$ donne $\langle x, x, x, \dots, x \rangle$

En reprenant le raisonnement que nous avons fait pour trouver la première expression de 2^n , on obtient

$*\text{mult } o [1^-, f]$

qui a pour résultat la séquence $\langle 1, x, \dots, x^n \rangle$ et

$\text{last } o (*\text{mult } o [1^-, f])$

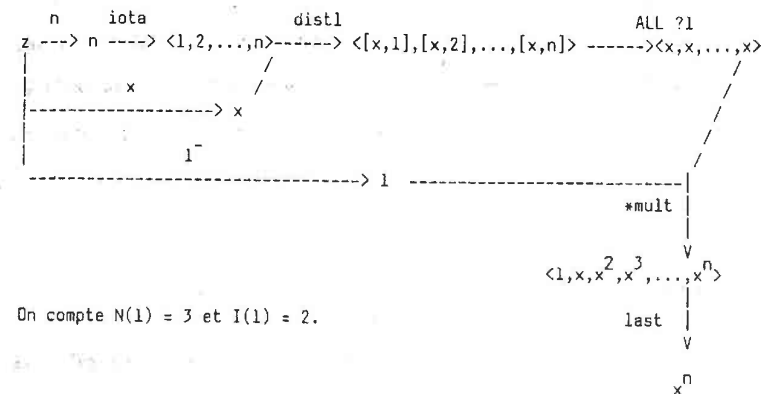
a pour résultat x^n .

première partie

L'EXPRESSION TROUVEE EST

(1) $\text{last } o (*\text{mult } o [1^-, \text{ALL ?1 } o (\text{distl } o [x, \text{iota } o n])])$

Le schéma de calcul est le suivant:



On compte $N(1) = 3$ et $I(1) = 2$.

● AVEC UN RAISONNEMENT DESCENDANT

Le raisonnement à effectuer est le même que pour obtenir la deuxième expression de 2^n . La difficulté supplémentaire vient de l'argument x .

On veut obtenir la séquence $s: \langle 1, x, x^2, \dots, x^n \rangle$; x^n en est le dernier terme.

La séquence s est résultat d'une itération " $*g$ ". Si le premier argument de g est x^{i-1} , g doit retourner x^i et ceci pour i de 1 à n . L'initialisation est alors x^0 .

Il est alors normal d'écrire l'expression

première partie

(e) $*g \circ [1^-, \text{iota} \circ n]$

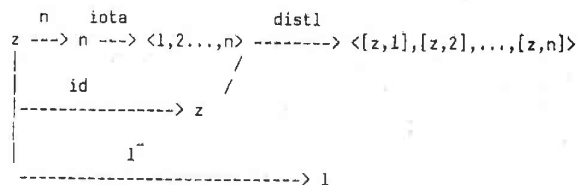
Il reste à écrire une expression de g . Pour i de 1 à n , g s'applique au couple $[x^{i-1}, i]$. On sait donc que "?1" donne x^{i-1} . Il suffit donc de multiplier par x . Il faut utiliser la fonction x pour obtenir cette valeur. Il est cependant faux de définir g par l'expression " $\text{mult} \circ [?1, x]$ ". En effet, x doit avoir pour argument un objet z argument de (e) et non un des couples $[x^{i-1}, i]$ argument de g . Pour obtenir cet x , il faut modifier l'argument de g . Il ne serait pas opportun de toucher au premier élément du couple qui donne les termes x^i de la séquence résultat. Le plus simple est de modifier le deuxième élément i .

PREMIERE SOLUTION

On modifie i en $[z, i]$. L'argument de " $*g$ " doit alors être le résultat de

(a) $[1^-, \text{distl} \circ [\text{id}, \text{iota} \circ n]]$

Le schéma suivant montre un calcul effectué par (a):



Au niveau de g , on obtient z par le sélecteur "?1o?2". g peut se définir par:

$\text{def } g = \text{mult} \circ [?1, x \circ (?1o?2)]$

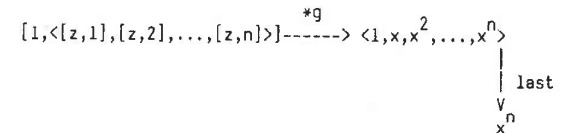
première partie

L'EXPRESSION EST ALORS

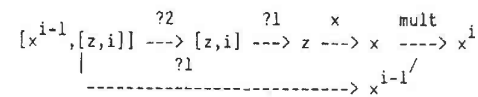
(2) $\text{last} \circ (*g \circ [1^-, \text{distl} \circ [\text{id}, \text{iota} \circ n]])$

where $\text{def } g = \text{mult} \circ [?1, x \circ (?1o?2)]$

Pour visualiser le calcul, nous complétons le schéma de (a) par:



avec, pour i de 1 à n , un calcul de g :



On compte $N(2) = 2$ et $I(2) = 1$.

Cependant, dans l'expression (2), x est placé dans l'expression de g . L'expression (2) spécifie n calculs de x pour notre interprétation. Cette expression est obtenue par un raisonnement structuré de manière descendante car on pense à x au moment où l'on en a besoin pour définir la fonction itérée g . La fonction x est alors placée naturellement dans l'expression de g . C'est en faisant un tel raisonnement que des débutants écrivent des programmes qui relisent n fois la donnée x .

AUTRE SOLUTION

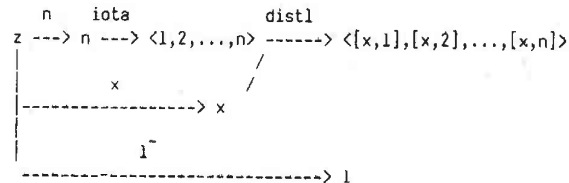
évite de placer x dans l'expression de g dans " $*g$ ". L'argument de " $*g$ "

première partie

doit alors être le résultat de

$$(b) \quad [1^-, \text{distl o } [x, \text{iota o } n]]$$

Le schéma suivant montre un calcul effectué par (b):



Au niveau de g' , on obtient x par le sélecteur "?1o?2" et g' se définit alors par

$$\text{def } g' = \text{mult o } [?1, ?1o?2]$$

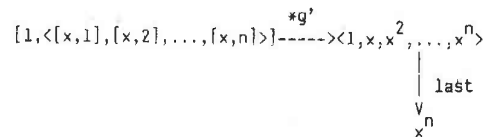
L'EXPRESSION TROUVÉE EST ALORS

$$(3) \quad \text{last o } (*g' \text{ o } [1^-, \text{distl o } [x, \text{iota o } n]])$$

$$\text{where def } g' = \text{mult o } [?1, ?1o?2]$$

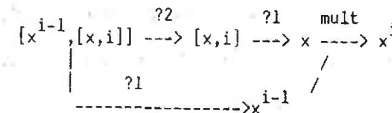
Cette fois, x est placé au bon endroit (hors de l'itération $*g'$).

On peut compléter le schéma ci-dessus par:



première partie

avec, pour i de 1 à n un calcul de g'



On compte $N(3) = 2$ et $I(3) = 1$. Les expressions (2) et (3) engendrent une séquence intermédiaire de moins que l'expression (1) trouvée selon une méthode ascendante. Mais elles sont moins lisibles.

4.2. Quand l'utilisation d'expressions connues ne donne pas une bonne solution

Nous proposons des expressions pour obtenir $2^n/n!$.

Certaines demandent deux itérations séparées, l'une donnant 2^n et l'autre donnant $n!$. Ces expressions spécifient alors deux séquences intermédiaires, celle des puissances successives de 2 et la séquence des $i!$ pour i de 0 à n .

Mais d'autres expressions n'exigent qu'une séquence intermédiaire. On peut proposer une expression qui correspond à un calcul parallèle de 2^n et $n!$. Cette expression construit la séquence intermédiaire des couples

$$[2^i, i!] \text{ pour } i \text{ de } 0 \text{ à } n.$$

Une meilleure expression prend le dernier terme de la séquence des $2^i/i!$ pour i de 0 à n .

première partie

On peut raisonner de manière analogue pour trouver des expressions qui correspondent à divers calculs de $\Sigma a_i b_i + \Sigma c_i d_i$. Supposons que l'on veuille réutiliser une expression de calcul du produit scalaire, on écrit alors deux itérations séparées, l'une calculant le produit scalaire $\Sigma a_i b_i$, l'autre calculant le produit scalaire $\Sigma c_i d_i$.

L'exemple est cité par R. BURSTALL et J. DARLINGTON [Bur77]. Leur système transforme le programme qui calcule séparément $\Sigma a_i b_i$ et $\Sigma c_i d_i$ en un programme qui calcule la somme

$$0 + (a_1 b_1 + c_1 d_1) + (a_2 b_2 + c_2 d_2) + \dots + (a_n b_n + c_n d_n)$$

Ce calcul est effectué par une seule itération qui cumule les "aibi+cidi" pour i de 1 à n.

a) Exemple 3

● POUR UNE EXPRESSION CALCULANT $2^n/n!$, ON UTILISE DES EXPRESSION DE 2^n ET $n!$

Nous pouvons reprendre les expressions déjà trouvées au paragraphe 3.1 qui ont pour résultat la séquence $\langle 1, 2, 4, 8, \dots, 2^n \rangle$

def u = *mult o [1⁻, ALL 2⁻ o (iota o n)]

ou encore

def u = *(mult o [2¹, 2⁻]) o [1⁻, iota o n].

Nous pouvons aussi écrire une fonction v qui a pour résultat la séquence $\langle 1, 1, 2, 6, \dots, n! \rangle$

première partie

def v = *mult o [1⁻, iota o n].

Les deux séquences obtenues peuvent alors être couplées par trans, "trans o [u,v]" donnant $\langle [1,1], [2,1], [4,2], [8,6], \dots, [2^n, n!] \rangle$, pour être "consommées en parallèle" par une itération "ALL /".

ALL / o (trans o [u,v])

a pour résultat $\langle 1, 2, 4/2, 8/6, \dots, 2^n/n! \rangle$.

Alors 2^n est résultat de

last o (ALL / o (trans o [u,v])).

Mais nous pouvons aussi dire que "last o (trans o [u,v])" a pour résultat $[2^n, n!]$ et donc que

"/ o (last o (trans o [u,v]))" a pour résultat $2^n/n!$

NOUS AVONS OBTENU LES EXPRESSIONS

(1) last o (ALL / o (trans o [u,v]))

ou

(2) / o (last o (trans o [u,v]))

where def u = *mult o [1⁻, ALL 2⁻ o (iota o n)]

ou encore

def u = *(mult o [2¹, 2⁻]) o [1⁻, iota o n]

et

def v = *mult o [1⁻, iota o n]

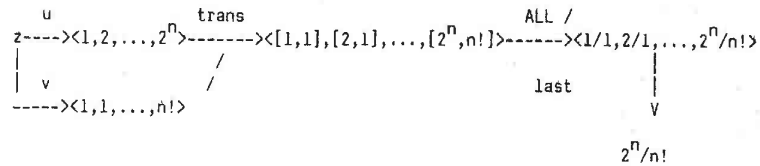
On peut aussi penser à utiliser des expressions de "last o u" pour avoir $n!$ et de "last o v" pour avoir 2^n et l'on trouve alors que $2^n/n!$ est résultat de l'expression

première partie

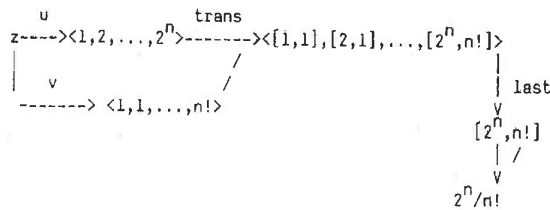
(3) / o [last o u, last o v]

Les schémas de calcul pour ces expressions sont

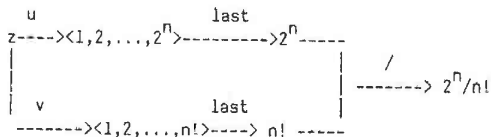
pour (1)



pour (2)



pour (3)



On peut compter:

$$N(1) = N(u) + N(v) + 2 \text{ et } I(1) = I(u) + I(v) + 1$$

$$N(2) = N(u) + N(v) + 1 \text{ et } I(2) = I(u) + I(v)$$

$$N(3) = N(u) + N(v) \text{ et } I(3) = I(u) + I(v)$$

● PAR UN RAISONNEMENT DESCENDANT

Avec une fonction f qui calcule $\langle 1, 1, 2, \dots, 2^n/n! \rangle$, "last o f" a pour résultat $2^n/n!$.

première partie

f est une itération "g". Si le premier argument de g est $2^{i-1}/(i-1)!$, g doit avoir pour résultat $2^i/i!$ et ceci pour i de 1 à n.

L'initialisation est alors $2^0/0!$, soit 1. On définit alors f

where def f = *g o [1⁻, iota o n]

Quand on écrit g, "?1" donne $2^{i-1}/(i-1)!$, il suffit donc que g multiplie par 2/i son premier argument. L'entier i est un terme de la séquence "iota o n". On trouve alors

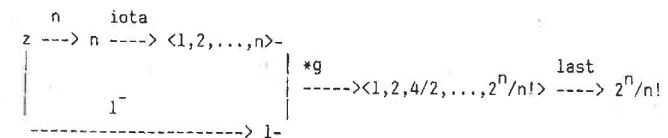
where def g = mult o [?1, / o [2⁻, ?2]]

L'EXPRESSION EST

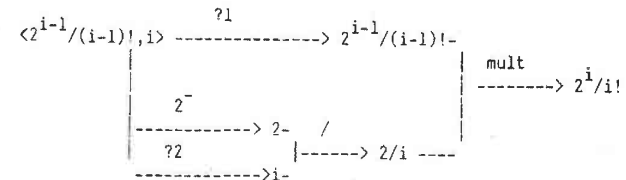
(4) last o (*g o [1⁻, iota o n])

where def g = mult o [?1, / o [2⁻, ?2]]

Le schéma du calcul est



avec les n calculs de g pour i de 1 à n



On compte $N(4) = I(4) = 1$.

première partie


```
def ps = last o (*+ o [0~,ALL mult o trans])
ou encore
def ps = last o (*(+ o [?1,mult o ?2]) o [0~,trans])
```

On peut introduire des fonctions a, b, c et d telles que, pour i de 1 à n, "a:i" soit l'élément ai, "b:i" soit l'élément bi, "c:i" soit l'élément ci et "d:i" soit l'élément di. Une fonction n donne la longueur commune des quatre vecteurs. On a alors

"ALL a o (iota o n)" qui donne <a1,a2,...,an>,
 "ALL b o (iota o n)" qui donne <b1,b2,...,bn>,
 "ALL c o (iota o n)" qui donne <c1,c2,...,cn> et
 "ALL d o (iota o n)" qui donne <d1,d2,...,dn>.

L'expression

```
ps o [ALL a o (iota o n),ALL b o (iota o n)]
```

a pour résultat le produit scalaire a1b1+a2b2+...+anbn.

L'expression

```
ps o [ALL c o (iota o n),ALL d o (iota o n)]
```

a pour résultat le produit scalaire c1d1+c2d2+...+cndn

L'EXPRESSION S'ECRIT ALORS

```
(1) + o [ps o [ALL a o (iota o n),ALL b o (iota o n)],
ps o [ALL c o (iota o n),ALL d o (iota o n)]]
```

Notre système de transformation sera capable de transformer cette expression (1) en une expression

première partie

```
(2) last o (*f o [0~,iota o n])
where def f = + o [?1
, + o [ mult o [a o ?2,b o ?2]
, mult o [c o ?2,d o ?2] ] ]
```

qui cumule les calculs parallèles des aibi et des cidi pour i de 1 à n.

Le système de R. BURSTALL et J. DARLINGTON demande un "eureka" (demande d'aide à l'utilisateur) pour faire cette transformation [Bur77].

4.3. A propos du raisonnement descendant

Pour les exemples que nous avons considérés jusqu'à maintenant, il peut sembler qu'un raisonnement descendant conduit à trouver des expressions qui spécifient moins de séquences intermédiaires que si l'on choisit un raisonnement ascendant. Cependant, ce n'est pas toujours le cas pour les exemples que nous allons étudier maintenant.

a) Exemple 5

● PROBLEME D'IMBRICATION

Nous voulons par exemple une expression calculant $1+2+4+\dots+2^n$. Nous utilisons une fonction de nom n donnant l'entier n. Une itération peut cumuler les puissances successives de 2, ce que fait "g", si, pour i de 1 à n, g ajoute 2^i à son premier argument.

L'expression s'écrit alors

```
last o (*g o [0~,iota o n])
```

première partie

Si une fonction f , appliquée à un argument i , donne 2^i , on peut écrire

def $g = + o [?1, fo?2]$

Pour définir f , il est alors possible de penser à écrire une expression du calcul de 2^i . Soit par exemple

where def $f = \text{last } o (*\text{mult } o [1^-, \text{ALL } 2^- o \text{iota}])$

L'EXPRESSION CONSTRUITE EST

(i) $\text{last } o (*g o [0^-, \text{iota } o n])$

where def $g = + o [?1, f o ?2]$

and def $f = \text{last } o (*\text{mult } o [1^-, \text{ALL } 2^- o \text{iota}])$

Cette expression effectue le calcul suivant

$$\begin{array}{c} \begin{array}{c} ?1 \\ 0+1+2+\dots+2^{i-1}, i \end{array} \xrightarrow{\quad f \quad} \begin{array}{c} 0+1+2+\dots+2^{i-1} \\ \hline \end{array} \xrightarrow{\quad + \quad} (0+1+2+\dots+2^{i-1})+2^i \\ \hline \xrightarrow{\quad \quad \quad} 2^i \end{array}$$

avec f qui, pour chaque i , calcule 2^i en effectuant

$$(((\dots(((1)2)2)\dots)2)$$

ce qui est pour le moins inefficace.

On compte $I(i) = 1 + n \cdot I(f)$ et $N(i) = 1 + n \cdot N(f)$ avec $I(f) = N(f) = 2$.

Un tel raisonnement conduit à écrire en PASCAL deux itérations "imbriquées"

première partie

```
S:=0;
FOR I :=1 TO N DO
  BEGIN
    P:=1;
    FOR J:= 1 TO I DO P:=P*2;
    S:=S+P
  END;
```

b) Exemple 6

QUAND UN CALCUL CONSTANT EST PLACE DANS UNE ITERATION

C'est un problème bien connu des programmes itératifs. On enseigne qu'il ne faut pas faire répéter un calcul "constant" (qui donne la même valeur pour tout i) par une itération d'indice i .

Un exemple nous est donné par le problème suivant : L'expression doit avoir pour résultat une séquence de n moyennes. Chaque moyenne est faite pour les n éléments d'une séquence de 10 valeurs avec les pondérations " $p_1, p_2, \dots, p_{10}$ ".

On appelle p une fonction telle que " $p:j$ " soit l'élément de rang j d'un vecteur de pondérations. Une fonction x appliquée au couple $[i,j]$ donne l'élément j du i -ème vecteur de valeurs. La fonction nommée n donne l'entier n .

● NOUS PROPOSONS UNE EXPRESSION COUTEUSE QUI ITERE LE CALCUL DE LA SOMME DES PONDERATIONS

Si, pour i de 1 à n , " $f:i$ " donne la i -ème moyenne; " $\text{ALL } f o (\text{iota } o n)$ " a pour résultat la séquence des n moyennes.

Avec " $s_1:i$ " donnant le i -ème produit vectoriel entre valeurs et pondérations et " $s_2:i$ " donnant la somme des pondérations, on peut définir f

première partie

par

where def f = / o [s₁, s₂]

Il suffit d'écrire des expressions de s₁ et de s₂. Nous débutons par s₂.

s₂ DOIT AVOIR POUR RESULTAT "p₁+p₂+...+p₁₀".

On écrit

where def s₂ = last o (** o [0⁻, u])

and def u = ALL p o (iota o 10⁻)

ou encore

def s₂ = last o (*(+ o [?1, p o ?2]) o [0⁻, iota o 10⁻])

ECRIVONS MAINTENANT UNE EXPRESSION DE s₁.

Appliquée à un entier j de 1 à 10, s₁ doit avoir pour résultat

$\langle p_1 x_{i1}, p_2 x_{i2}, \dots, p_{10} x_{i10} \rangle$

s₁ est l'expression du dernier terme de la séquence résultat d'une itération " *g ". g ajoute le j-ème produit " p_j x_{ij} " et ceci pour j de 1 à 10. Si u donne la séquence des pondérations et v celle des 10 valeurs x_{ij}, on peut écrire

where def s₁ = last o (** o [0⁻, ALL mult o (trans o [u, v])])

u a pour résultat $\langle p_1, p_2, \dots, p_{10} \rangle$, on a déjà défini pour s₂

def u = ALL p o (iota o 10⁻)

première partie

Il reste v qui doit fournir $\langle x_{i1}, x_{i2}, \dots, x_{i10} \rangle$. Il nous faut pour cela la séquence

$\langle [i, 1], [i, 2], \dots, [i, 10] \rangle$

résultat de "distl o [id, iota o 10⁻]"

et alors

def v = ALL x o (distl o [id, iota o 10⁻])

L'EXPRESSION DE s₁ EST FINALEMENT

where def s₁ =

last o (** o [0⁻,

ALL mult o (trans

o [ALL p o (iota o 10⁻),

ALL x o (distl o [id, iota o 10⁻])])

L'EXPRESSION DONNANT LA SEQUENCE DES N MOYENNES

(1) ALL (/ o [s₁, s₂]) o (iota o n)

recommence le calcul de la somme des pondérations s₂ pour chaque calcul d'une des n moyennes. Cette aberration se produit quand on raisonne sur un mode descendant car on place s₂ à l'endroit où l'on veut s'en servir, c'est à dire au moment de calculer une des moyennes.

On peut d'ailleurs compter

$N(1) = n * (N(s_1) + N(s_2)) + 1$ et $I(1) = n * (I(s_1) + I(s_2)) + 1$.

● MEILLEURES EXPRESSIONS

première partie

Il est pourtant plus simple d'écrire une expression fonctionnelle qui utilise pour chaque calcul d'une moyenne la somme des pondérations, résultat de s_2 . Elle peut être

$$(2) \text{ ALL } (/ \text{ o } [?2, ?1]) \text{ o } (\text{dist1 o } [s_2, \text{ALL } s_1 \text{ o } (\text{iota o } n)])$$

qui engendre la séquence intermédiaire des n moyennes avec la fonction

$$\text{ALL } s_1 \text{ o } (\text{iota o } n).$$

L'expression suivante n'engendre pas cette séquence intermédiaire:

$$(3) \text{ ALL } (/ \text{ o } [s_1 \text{ o } ?2, ?1]) \text{ o } (\text{dist1 o } [s_2, \text{iota o } n])$$

On compte:

$$N(2) = n * N(s_1) + N(s_2) + 3 \quad \text{et} \quad I(2) = n * I(s_1) + I(s_2) + 2$$

$$N(3) = n * N(s_1) + N(s_2) + 2 \quad \text{et} \quad I(3) = n * I(s_1) + I(s_2) + 1.$$

$N(s_2)$ ou $I(s_2)$ ne sont plus multipliés par le coefficient n .

c) Exemple 7

QUAND ON NE CHOISIT PAS LA BONNE SEQUENCE INTERMEDIAIRE

Il peut arriver que l'on ne choisisse pas la bonne séquence intermédiaire. On calcule par exemple une moyenne de 10 entiers, $\langle x_1, x_2, \dots, x_{10} \rangle$, en la considérant comme le dernier terme d'une séquence de moyennes successives. Ce peut être la séquence

$$(a) \langle 0, x_1, (x_1+x_2)/2, (x_1+x_2+x_3)/3, \dots, (x_1+x_2+\dots+x_{10})/10 \rangle;$$

ou encore la séquence

$$(b) \langle 0, x_1/10, (x_1+x_2)/10, \dots, (x_1+x_2+\dots+x_{10})/10 \rangle;$$

première partie

au lieu de diviser par 10 le dernier terme de la séquence suivante

$$(c) \langle 0, x_1, x_1+x_2, \dots, x_1+x_2+\dots+x_{10} \rangle.$$

Appelons x une fonction donnant pour résultat la séquence

$$\langle x_1, x_2, \dots, x_{10} \rangle.$$

● DANS LE CAS OU L'ON PENSE A LA SEQUENCE (b)

La moyenne est le dernier terme de (b). Pour obtenir (b), on utilise une itération $"*f"$. f doit diviser par 10 le cumul $"x_1+x_2+\dots+x_i"$. Le cumul peut être le résultat d'une fonction g . g ajoute x_i à $"x_1+x_2+\dots+x_{i-1}"$. Le deuxième argument de f doit être i . f est alors définie par

$$\text{def } f = [/ \text{ o } [g, 10^-], g]$$

ayant pour résultat le couple $"[(x_1+x_2+\dots+x_i)/i, x_1+x_2+\dots+x_i]"$. g s'applique à $"x_1+x_2+\dots+x_i"$ obtenu par $"?2o?1"$ et à $"x_i"$ obtenu par $"?2"$. On a

$$\text{def } g = + \text{ o } [?2o?1, ?2]$$

$"*f"$ a alors pour résultat non pas la séquence (b) mais la séquence

$$\langle [0, 0], [x_1/10, x_1], [(x_1+x_2)/10, x_1+x_2], \dots, [(x_1+x_2+\dots+x_{10})/10, x_1+x_2+\dots+x_{10}] \rangle$$

Un premier argument de $"*f"$ est l'initialisation $[0, 0]$ résultat de

$$[0^-, 0^-]$$

Un deuxième argument doit être la séquence

$$\langle x_1, x_2, \dots, x_{10} \rangle \text{ résultat de la fonction } x.$$

première partie

LA MOYENNE EST ALORS

(1) ?1 o (last o *f o [[0⁻,0⁻],x])

where def f = [/ o [g,10⁻] , g]

and def g =+ o [??o?1,?2]

● EN PARTANT DU CAS DE LA SEQUENCE (a),

la séquence des cumuls

$\langle 0, x_1, x_1+x_2, \dots, x_1+x_2+\dots+x_n \rangle$

est résultat de "++ o [0⁻,x]".

On peut alors penser à obtenir (a) en divisant chaque terme de

$\langle [x_1,1], [x_1+x_2,2], \dots, [x_1+x_2+\dots+x_n,n] \rangle$

Cette séquence est obtenue par

def u = trans o [tl o (++ o [0⁻,x]),iota o 10⁻]

on a alors

(2) last o (ALL / o u)

● EN DIVISANT PAR 10 LE DERNIER TERME DE LA SEQUENCE (c) DES CUMULS DES x_i

on obtient

(3) / o [last o (*+ o [0⁻,x] , 10⁻]

On compte:

première partie

N(1) = 1 et I(1) = 1

N(2) = 4 et I(2) = 2

N(3) = 1 et I(3) = 1

L'expression (1) est cependant moins bonne que l'expression (3): Dans "f", on itère l'opération "/". "f" est plus coûteuse que "+".

4.4. A propos des conditionnelles

Il arrive le plus fréquemment d'écrire des tests superflus parce que

a) Exemple 8

● ON RECOMMENCE UN TEST DÉJÀ FAIT

En raisonnant sur un mode descendant, il peut arriver d'oublier un test déjà effectué et de le recommencer.

Il faut par exemple obtenir une séquence de libellés au sujet de n moyennes.

Nous appellerons "moy" une fonction ayant pour résultat la moyenne. "moy" peut être définie par une des expressions trouvées pour l'exemple 6.

Avec les tests

Let def p1 = gt o [moy,12⁻]

and def p2 = gt o [moy,10⁻]

and def p3 = gt o [moy,8⁻]

on écrit l'expression

(e) p1 -> r ; (p2 -> s ; (p3 -> t ; u))

Première partie

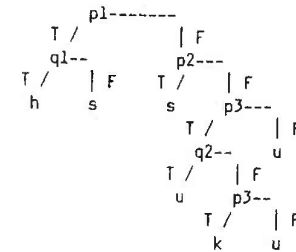
avec des fonctions r,s,t,u qui ont pour résultats respectifs les libellés appropriés aux quatre cas d'une moyenne supérieure à 12, d'une moyenne comprise entre 10 et 12, d'une moyenne comprise entre 8 et 10 et d'une moyenne inférieure à 8.

Pour les fonctions r et t, le libellé va dépendre du (caractère "a" ou "b") résultat d'une fonction num.

On a par exemple

```
where def r = q1 -> h ; s
      where def q1 = eq o [num,a~]
      and def h = [num,moy,mention bien~]
      and def s = [num,moy,admis~]
and def u = [num,moy,recalé~]
and def t = q2 -> u ; (p3 -> k ; z)
      where def q2 = eq o [num,b~]
      and def k = [num,moy,oral~]
      and def z = [num,moy,recalé~])
```

Dans l'expression de t, on recommence inutilement un test effectué par le prédicat p3. En effet t est placée dans l'expression (1) là où le prédicat p3 a toujours un résultat "T". La redondance du test p3 peut se voir sur un organigramme



b) Exemple 9

● AUTRE RÉDONDANCE DE TEST

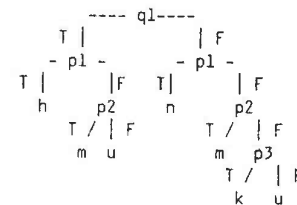
Pour un problème analogue, on écrit une expression

```
(f) q1 -> (p1 -> h ; ( p2 -> m ; u ))
      ; (p1 -> n ; ( p2 -> m ; (p3 -> k ; u)))
```

Les fonctions q1, p1, p2, p3, u, h et k sont celles de l'exemple 8. m et n s'écrivent

```
where def m = [num,moy,mention passable~]
and def n = [num,moy,mention tres bien~]
```

Dans le cas où le résultat de f est m, l'organigramme montre que l'on teste inutilement q1



4.5. Exemple 10. Il faut connaître des propriétés des fonctions qu'on utilise

Pour obtenir divers types de données dans une expression fonctionnelle, il suffit d'introduire les fonctions appropriées. Ces primitives ont des propriétés qui peuvent servir à écrire une bonne expression ou à transformer les expressions. Nous donnons un petit exemple.

Soit une fonction "inv" telle que "inv:[t,i]" échange les éléments de rang i et i+1 dans un vecteur t,

soit une fonction "decal" telle que "decal:[t,i]" place l'élément de rang i+1 au rang i,

soit une fonction "val" telle que "val:[t,i]" donne le i-ème élément de t,

soit enfin une fonction "place" telle que "place:[x,t,i]" met la valeur x au rang i dans t.

Appelons t une fonction donnant la suite t.

LES EXPRESSIONS

```
(1) last o (*inv o [t,iota o n])
et (2) place o [val o [t,1~]
      ,last o (*decal o [ t ,iota o n ])
      , + o [1~,n] ]
```

décalent les éléments de rang 2,3,...n+1 de t respectivement aux rangs 1,2,...n et placent celui de rang 1 au rang n+1. C'est ce que fait expli-

première partie

citement l'expression (2). L'expression (1) arrive au même résultat par une suite de n inversions.

On peut préférer la deuxième expression si l'on connaît des propriétés des fonctions "inv", "decal", "place" et "val"

Par exemple, pour tout vecteur t à tout rang i dans t, on a

```
inv:[t,i] == place:[val:[t,i],decal:[t,i],i+1].
```

Cette propriété s'exprime au niveau fonctionnel par

```
(PR1) inv = place o [val,decal,+ o [1~,?2]]
```

En plus de PR1, il faut connaître d'autres propriétés pour justifier que les définitions (1) et (2) donnent la même fonction. Mais, on peut se servir de PR1 pour constater que la fonction "inv" est plus coûteuse que "decal". Ainsi, on choisira plutôt l'expression (2).

4.6. Un dernier exemple

Les exemples précédents servent à introduire le but de notre travail. Nous considérons des expressions correctes construites selon un mode de raisonnement "avec séquences". Nous avons vu que cela conduisait à obtenir des expressions qui multiplient les séquences intermédiaires (exemples 1, 2, 3, 4, 5, 6 et 7) et introduisent des redondances de test (exemples 8 et 9). L'expression la plus simple n'est pas non plus toujours la meilleure quand on considère de manière fine les propriétés des fonctions qu'on emploie (exemple 10). C'est pourquoi nous voulons obtenir un système de transformation qui améliore ces expressions.

première partie

Terminons par l'exemple qui nous a donné l'idée de ce travail

Il s'agissait de décrire le calcul d'une exponentielle par sa série. Des étudiants utilisant MEDEE (le mode de construction de programmes structurés que nous leur proposons: [Pai79], [Fin79], [Guy84]) me suggéraient alors quelquefois le programme PASCAL suivant:

```
PROGRAM EXPO;
VAR
  EPSI,X,S,U,V,T:REAL;
  I,J:INTEGER;
BEGIN
  READ(X,EPSI);
  S:=1;I:=0;T:=1;
  WHILE (ABS(T)>EPSI) OR (I < 100) DO
  BEGIN
    I:=I+1;
    %calcul de xi%
    U:=1;
    FOR J :=1 TO I DO U:=U*X
    %calcul de i!%
    V:=1;
    FOR J:=1 TO I DO V:=V*J;
    T:= U/V;
    S:=S+T
  END;
  WRITE(S)
END.
```

Les calculs de $i!$ et de x^i sont recommencés pour chaque i (comme pour l'exemple 5). Les deux itérations consommant la même séquence $\langle 1,2,\dots,i \rangle$ sont faites séparément et les intermédiaires u et v sont inutiles (comme dans les expressions (1) (2) et (3) de l'exemple 3).

Pour comprendre le raisonnement conduisant à écrire un tel programme, nous allons écrire une expression fonctionnelle ayant pour résultat

$$1 + x + x^2/2! + \dots + x^{100}/100!$$

première partie

Pour simplifier, nous ne comparons pas le terme général de la série avec un seuil donné mais le raisonnement serait identique, il suffirait d'utiliser "WH".

CONSTRUCTION D'UNE EXPRESSION CALCULANT LA SOMME $1+x+\dots+x^{100}/100!$

Nous appelons x une fonction donnant le réel x . Si l'on définit une fonction f qui cumule les $x^i/i!$, la somme est le résultat de l'expression "last o f".

Si des fonctions u et v ont pour résultat x^i et $i!$, "/ o [u,v]" a pour résultat $x^i/i!$.

Si "f" a pour résultat la séquence de triplets

$\langle [1,1,1], [1+x,x,1], [1+x+x^2/2!,x^2,2!], \dots, [1+x+\dots+x^{100}/100!,x^{100},100!] \rangle$

f peut s'écrire

let def $f = [+ o [?lo?1, / o [u,v]] , u, v]$

L'expression

?1 o (last o *f)

donne alors la somme.

Il reste à écrire l'expression de l'argument de "f". On sait que

dist1 o [x,iota o 100~]

donne une séquence $\langle [x,1], [x,2], \dots, [x,100] \rangle$.

La valeur initiale est le triplet $[1,1,1]$. "f" peut s'appliquer au

première partie

résultat de l'expression

$[[1^-, 1^-, 1^-], \text{distl o } [x, \text{iota o } 100^-]]$

Nous pouvons alors définir v qui a pour résultat $i!$. Connaissant le deuxième élément argument de f : $[x, i]$, on voit que $"?2"$ donne $[x, i]$ et $"?2o?2"$ donne i . la fonction v est définie par

$\text{def } v = \text{last o } (*\text{mult o } [1^-, \text{iota o } (?2o?2)])$

Il reste à définir u qui a pour résultat x^i .

u est une itération $"\text{last o } *r"$. Si r a un premier argument x^{j-1} , r doit avoir pour résultat x^j , il suffit de multiplier par x .

Connaissant le deuxième élément argument de f : $[x, i]$, $"?1o?2"$ donne x et $"?2o?2"$ donne la puissance i de x . On écrit alors

$\text{def } u = \text{last o } (*r o [1^-, \text{distl o } [?1o?2, \text{iota o } (?2o?2)]])$

et $\text{def } r = \text{mult o } [?1, ?1o?2]$

L'EXPRESSION TROUVEE EST ALORS

(1)

$?1 o (\text{last o } (*f o [[1^-, 1^-, 1^-], \text{distl o } [x, \text{iota o } 100^-]]))$

where $\text{def } f = [+ o [?1o?1, / o [u, v]] , u , v]$

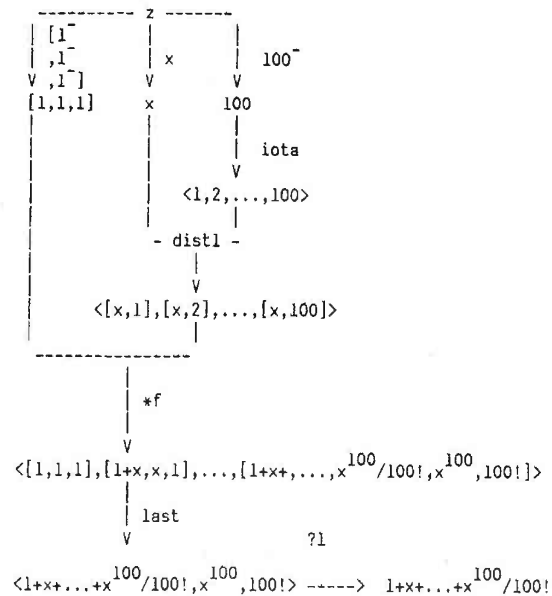
and $\text{def } v = \text{last o } (*\text{mult o } [1^-, \text{iota o } (?2o?2)])$

and $\text{def } u = \text{last o } (*(\text{mult o } [?1, ?1o?2])$

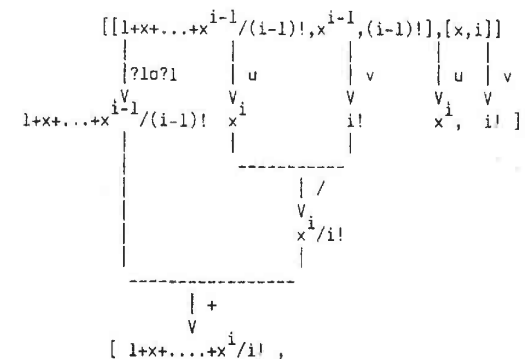
$o [1^-, \text{distl o } [?1o?2, \text{iota o } (?2o?2)]])$

Le calcul se fait selon le schéma suivant:

première partie



De plus, le calcul de $"*f"$ se fait selon le schéma



On compte le nombre de séquences intermédiaires

Première partie

$N(1) = 100 \cdot N(u) + 100 \cdot N(v) + 2$ avec $N(u) = 2$ et $N(v) = 1$

donc $N(1) = 302$

et le nombre d'itérations

$I(1) = 100 \cdot I(u) + 100 \cdot I(v) + 1$ avec $I(u) = 1$ et $I(v) = 1$

donc $I(1) = 201$

CONSTRUISONS DE MEILLEURES EXPRESSIONS

Pour

let def $f = \text{*mult o [1^-,iota o 100^-]}$,

f a pour résultat $\langle 0!, 1!, \dots, 100! \rangle$.

Pour

let def $g = \text{*mult o [1^-, ALL?1 o (dist1 o [x,iota o 100^-])]}$,

g a pour résultat $\langle 1, x, \dots, x^{100} \rangle$.

Donc

let def $k = \text{ALL / o (trans o [g,f])}$,

k a pour résultat $\langle 1/0!, x/1!, \dots, x^{100}/100! \rangle$.

L'expression

$\text{last o (*+ o [1^-,tl o k])}$

calcule la somme voulue.

L'EXPRESSION S'ECRIT

première partie

(2) $\text{last o (*+ o [1^-,tl o k])}$

where def $k = \text{ALL / o (trans o [f,g])}$

where def $f = \text{*mult o [1^-,iota o 100^-]}$

and def $g = \text{*mult o [1^-, ALL?1 o (dist1 o [x,iota o 100^-])]}$

On compte $N(2) = 7$ et $I(2) = 5$

MAIS ON PEUT TROUVER AUSSI L'EXPRESSION

(3)

$?1 o (\text{last o (*f o [[1^-,1^-],dist1 o [x,iota o 100^-]])})$

where def $f = + o [?1o?1, \text{mult o [?2o?1, / o [?1o?2, ?2o?2]]}]$

On compte $N(3) = 2$ et $I(3) = 1$

L'expression (3) correspond au programme PASCAL

```
PROGRAM SERIE;
VAR
  X,S,T:REAL;
  I:INTEGER;
BEGIN
  READ(X);
  S:=1;T:=1;
  FOR I:= 1 TO 100 DO
    BEGIN
      T:=T*X/I;
      S:=S+T
    END;
  WRITE(S);
END.
```

L'expression (3) est la meilleure pour le nombre des séquences intermédiaires.

5. Les expressions fonctionnelles

Maintenant, nous sommes en mesure de donner une définition complète des expressions fonctionnelles que nous utilisons pour coder les programmes.

première partie

5.1. Expression fonctionnelle

L'ensemble A des atomes est la réunion des booléens, des entiers, des réels, des caractères, des chaînes de caractères, de "!" et de "<>". Nous excluons toute notation d'objet qui n'est pas un atome. Les atomes sont les arguments de "_". Soit C l'ensemble des fonctions.

L'ensemble P des primitives comprend les primitives de base citées au paragraphe 2.3 et les primitives sur les séquences citées au paragraphe 2.4.

On pourra ajouter des fonctions particulières pour traiter des types de données différents des séquences. C'est ce que nous avons fait dans l'exemple 10 pour le type vecteur en introduisant les fonctions "decal", "inv", "place" et "val".

Notons aussi que l'ensemble P est infini car il contient les sélecteurs.

L'ensemble FF des formes fonctionnelles comprend

- a) les trois formes fonctionnelles de base {_o,[_..._],_->_},
- b) les trois formes fonctionnelles sur les séquences {ALL _,*_ ,WH_}.

Cet ensemble est volontairement réduit (cf. paragraphe 2.6.).

Remarquons aussi que [_..._] est d'arité multiple.

L'ensemble F des expressions fonctionnelles est alors défini récursivement

- a) C et P sont inclus dans F

première partie

b) si $f_1, f_2, \dots, f_n$ sont dans F et si E , d'arité n , est dans FF alors $E(f_1, f_2, \dots, f_n)$

est dans F

Ainsi nous n'autorisons pas la définition de J. BACKUS [Bac78] qui ajoute la possibilité de définir une fonction par une équation récursive

def $f = E(f)$ où $E(f)$ est une expression fonctionnelle contenant f .

Une fonction donnant $n!$ serait alors définie par l'expression récursive

def $f = \text{eq0} \rightarrow 1^- ; \text{mult } o \text{ [id, } f \text{ } o \text{ (} - \text{ } o \text{ [1}^-, \text{id})]}$

Nous n'avons pas besoin d'écrire ces définitions récursives.

Pour exprimer les fonctions, nous pouvons utiliser uniquement les formes fonctionnelles "ALL", "*" et "WH" que nous avons choisies. Ceci est théoriquement assez puissant et correspond mieux au style des programmes que nous considérons.

Nous décrivons par exemple la fonction f de résultat $n!$ par une "itération"

def $f = \text{last } o \text{ (*mult } o \text{ [1}^-, \text{iota } o \text{ } n])$

Les formes fonctionnelles "ALL", "*" et "WH" permettent de faire en sorte que l'ensemble des fonctions décrites par de telles expressions fonctionnelles soit équivalent à la classe des programmes itératifs qui se terminent. Nous ne transformons que des programmes corrects aussi les programmes qui bouclent ne nous intéressent pas. La définition de "WH" empêche de coder les programmes itératifs qui ne se terminent pas.

première partie

5.2. Les fonctions sont toutes strictes

Les ensembles P et FF sont tels que toutes les fonctions f de F soient strictes donc, pour toute fonction f dans F,

$$f: ! \Rightarrow !$$

J. WILLIAMS [Wil80] fait remarquer que la forme fonctionnelle " $_ \rightarrow _ ; _$ " n'est évidemment pas elle même une fonction stricte. Cependant les fonctions construites par " $_ \rightarrow _ ; _$ " sont des fonctions strictes. En effet, pour toutes fonctions p, f, g strictes, la fonction " $p \rightarrow f ; g$ " est stricte

$$(p \rightarrow f ; g): ! \Rightarrow (p: ! \rightarrow (f: !)) ; (g: ! \Rightarrow ! \rightarrow ! ; ! \Rightarrow !)$$

Le fait que toutes les fonctions FP soient strictes permet d'affirmer qu'une fonction f définie par une équation récursive "Def f = E(f)" est le plus petit point fixe solution de cette équation.

Bien que nous rejetions les définitions récursives dans notre restriction du langage FP, il faut que nous définissions les fonctions données par les formes fonctionnelles "ALL", "*" et "WH" ainsi que les primitives qui, elles, sont faites par des définitions récursives.

Pour toute fonction f et pour tout prédicat p, les fonctions " $*f$ " et " $WH(p, f)$ " ainsi que les primitives comme "last", "tl", ... sont les solutions d'équations récursives.

Appelons I une fonction " $*f$ ", il est possible de l'écrire avec une définition récursive:

première partie

```
def(I)= paire ->
  ((null o ?2) -> ?1
   ; ((define o (hd o ?2)) ->
      cons o [?1, I o [f o [?1, ?1o?2], tl o ?2]]
      ; !~))
  ; !~
```

Il suffit d'écrire les prédicats "paire", "null" et "define".

"paire" est un prédicat qui donne "T" quand il s'applique à un couple. "paire" permet de définir que " $*f$ " est d'arité 2.

"null" est un prédicat qui donne "T" quand il s'applique à la séquence vide. Il permet de définir " $*f$ " dans le cas " $*f : [z, < >]$ ".

"define" est un prédicat qui a pour résultat "T" quand il s'applique à un objet x différent de "!". Dans le cas où le sélecteur "hd o ?2" ne donne pas "!", c'est que son argument est une séquence d'au moins un élément (c'est à dire d'une séquence non vide). On est dans le cas " $*f : [z, cons: [x, a]]$ ".

Nous pouvons écrire des définitions récursives pour "ALL f", "WH(p, f)" et pour les fonctions primitives. Pour simplifier, nous avons donné des formes plus intuitives au paragraphe 2.

C'est pourquoi nous voulons aussi que toutes les fonctions soient strictes.

Une remarque importante est que tout sélecteur doit aussi être une fonction stricte. On doit avoir par exemple

$$?1 : [3, !] \Rightarrow ?1 : ! \Rightarrow !$$

première partie

5.3. Selecteurs et illisibilité des expressions FP

Les expressions fonctionnelles qu'on exprime "au niveau des fonctions" semblent souvent peu claires. L'illisibilité est due à la suppression des variables d'objets arguments de la fonction.

Ces arguments apparaissent sous forme de variables d'objets quand on écrit l'expression "au niveau des objets", ce qui permet de lire facilement de cette forme d'expression.

Les arguments sont remplacés par des sélecteurs dans l'expression fonctionnelle "au niveau des fonctions", ce qui entraîne l'hermétisme de cette forme d'expression.

Par exemple la fonction:

(1) $gt \rightarrow ?1 ; ?2$

donne le maximum de deux nombres "comparés" par "gt".

Cependant on peut aussi introduire deux fonctions x et y qui donnent les deux nombres et on écrit alors l'expression

(2) $gt \circ [x,y] \rightarrow x ; y$

qui donne le maximum des deux nombres résultats de x et de y.

L'expression (2) est plus lisible que l'expression (1). Nous avons souvent utilisé cette astuce dans les exemples du paragraphe 3. J. BACKUS appelle "expression pure" la formule (1) et "expression étendue" la formule (2) [Bac81a].

première partie

Cependant quand on écrit les fonctions f (et p) arguments de "*f", "ALL f" et "WH(p,f)" on utilise les sélecteurs pour obtenir les composantes de l'argument de f (et p). C'est ce qui gêne la lecture de nos expressions fonctionnelles.

Rappelons encore une fois que notre but n'est pas d'avoir un bon langage fonctionnel pour lire les programmes, mais de pouvoir coder simplement les programmes itératifs par des expressions fonctionnelles pour les transformer en espérant améliorer leur efficacité. Autrement dit: **nous voulons diminuer le nombre de séquences intermédiaires qu'elles engendrent.**

Aussi, nous abandonnons la vision des programmes comme des textes mais nous préférons les voir comme des expressions fonctionnelles et ainsi nous pourrions faire de l'"algèbre" sur les programmes.

6. Bibliographie

Au sujet de la construction structurée des programmes, l'ouvrage de référence est celui de E. W. DIJKSTRA [DiJ76].

L'utilisation des suites infinies pour construire les programmes est mise en évidence dans le langage LUCID ([Ash76],[Ash77]). Les idées d'une méthode de construction structurée des programmes itératifs utilisant des suites finies ont été proposées par C. PAIR [Pai79]. Une approche similaire a été faite par J. ARSAC [Ars77]. Elles ont permis de définir la méthode MEDEE [Bel78] qui a été utilisée dans l'enseignement de l'initiation à la programmation. Il existe un livre d'initiation à la programmation utilisant cette méthode [Duc84] ainsi qu'un éditeur de haut niveau pour manipuler des

première partie

expressions et des "textes" MEDEE [Guy84]. Il est possible de le traduire dans divers langages. En particulier, il serait facile de le traduire dans des expressions FP. Alors notre travail pourrait être utilisé à ce niveau dans le cadre de MEDEE.

Le premier langage fonctionnel est LISP. La base de LISP est le Lambda-calcul ([Bar81],[Ber81a]. Nous pouvons citer HOPE [Bur80], ML ([Car80],[Gor77]), SASL [TUR,76] et VAL [Ack79]. P. J. LANDIN a bien montré l'utilité d'un langage fonctionnel [Lan66] ainsi que [Tur81]. L'utilisation des langages fonctionnels pour la construction des programmes est expliquée dans les livres de W. H. BURGE [Bur75] et de P. HENDERSON [Hen80]. Le langage FP introduit par J. BACKUS ([Bac78], [Bac81a], [Bac81b]) se distingue en évitant de noter les fonctions sous leur forme applicative habituelle du Lambda-calcul " $f(x) = \lambda x.e$ " mais par une formule qui utilise des opérations sur les fonctions comme l'opération de composition "o". C'est ce qui permet d'utiliser facilement une algèbre des fonctions [Bac78], [Wil80].

Nous avons constaté que U.S. REDDY utilisait le langage FP pour mettre en évidence l'utilisation des séquences (suites finies) dans les programmes itératifs [Red82]. Cela lui permet de détecter les formules récursives qui correspondent à des programmes itératifs [Jay83].

Dans le langage APL, les formules sur les "vecteurs" ressemblent aux formules sur les séquences qui codent en FP les programmes itératifs. Si ces formules sont interprétées, sans modification, à la manière d'une machine de VON NEUMANN, c'est terriblement inefficace. Un bon compilateur APL optimise ces formules. Un travail sur l'optimisation des programmes

APL est [Gui78].

Dans le but d'éclaircir les principes de base des améliorations à effectuer sur ces programmes itératifs, nous transformons la formule codant le programme (transformation du programme source), ce qui distingue notre travail de ceux que nous avons cités ci-dessus. Quand on écrit un compilateur, on peut utiliser les structures de données inhérentes au compilateur pour optimiser ce qui cache la nature de la transformation.

La plupart des travaux sur la transformation d'un programme source en un autre programme source concernent la transformation des programmes récursifs en programmes itératifs. Le travail le plus important dans ce domaine est le système de R. BURSTALL et J. DARLINGTON ([Bur77], [Fea79]) Mais nous voulons seulement améliorer la formulation des programmes itératifs [Bel80].

Avec un objectif similaire au notre, J.P. FINANCE et J. SOUQUIERES ont décrit un système de transformation des programmes MEDEE ([Fin85],[Sou85]). Ils utilisent un ensemble de règles de transformations de même que B. WEGBREIT [Weg76] et F.L. BAUER, M. BROU et dans le projet CIP ([Bau79a],[Bau79b]). D'autre part, G. HUET et B. LANG ont proposé une technique de transformations de programmes basée sur la réécriture de schémas de second ordre [Hue78].

Notre intention est de partir d'expressions FP codant des programmes itératifs. Notre projet est moins ambitieux que celui de P. WADLER [Wad81], qui transforme des expressions fonctionnelles avec des formes fonctionnelles engendrant des fonctions sur les listes (ce qui peut être assimilé à nos séquences uniquement quand les listes sont finies). Les

expressions peuvent être récursives, ce que nous excluons.

Nous ne supprimerons pas non plus toutes les séquences intermédiaires mais uniquement celles qui sont introduites lors de la construction méthodique du programme. P. WADLER [Wad84] supprime toutes les séquences intermédiaires des programmes évaluables sur un nombre borné d'emplacements par le moyen dit de "listleness evaluation" qui est dérivé de la "lazy evaluation" [Hen76].

THEORIE EQUATIONNELLE VALIDEE PAR L'ALGEBRE DES EXPRESSIONS FONCTIONNELLES

Les expressions fonctionnelles constituent une algèbre: la FP- algèbre. Si nous appelons V un ensemble de variables de fonction. Cela permet de considérer les expressions fonctionnelles comme des termes d'une algèbre libre qui constituent l'ensemble des FP-termes. La description de la FP- algèbre et des FP-termes font l'objet du paragraphe 1.

L'égalité que nous voulons décrire dans la FP-algèbre est l'égalité extensionnelle autrement dit nous allons nous intéresser aux équations validées par la FP-algèbre. Nous parlerons d'équations mais aussi d'équations conditionnelles au paragraphe 2.

Considérant un ensemble A d'équations validées par la FP-algèbre, un modèle de A est un quotient de l'algèbre des FP-termes et s'envoie par morphisme dans la FP-algèbre. Il est alors possible de considérer la théorie équationnelle engendrée par la A -égalité. En vertu du théorème de BIRKOFF [Bir35], on peut, dans une théorie équationnelle, prouver la validité d'une équation en utilisant uniquement les règles syntaxiques de la logique équationnelle (substitution et remplacement d'un égal par un égal). Remarquons que toute équation valide dans la théorie équationnelle l'est alors aussi dans la FP-algèbre. Il est alors envisageable d'automatiser les preuves d'équivalence et les transformations dans cette théorie équationnelle.

A notre connaissance, il n'existe pas une théorie équationnelle engendrée par un nombre fini d'équations dont la FP-algèbre soit le modèle initial. Si nous possédons diverses expressions $e_1, e_2, \dots, e_n$ d'une même fonction ($e_1, e_2, \dots, e_n$ ont été trouvées par n raisonnements structurés

différents comme pour les exemples du chapitre I), nous voulons simplement trouver un ensemble d'équations A tel que la A-égalité suffise pour prouver que $e_1, e_2, \dots, e_n$ sont équivalentes. Nous voulons que notre ensemble A nous permette de prouver la A-égalité des expressions différentes d'une même fonction. En particulier, nous voulons pouvoir prouver que les diverses expressions d'une même fonction que nous avons trouvées pour chacun des exemples du chapitre I sont A-égales.

Plus exactement, la A-égalité est une congruence qui partage les expressions en classes d'équivalence. Notre but est qu'il existe dans chaque classe une expression qui soit optimum pour le nombre de séquences intermédiaires.

Nous définirons notre ensemble A au paragraphe 3.

Cependant, nous constaterons que l'ensemble A ne nous permet pas d'atteindre tout à fait notre objectif. Certains cas nécessitent l'application d'équations valides dans la FP-algèbre qui ne sont pas conséquences de A. Nous donnerons des exemples au paragraphe 4 et aussi des mécanismes d'inférence qui peuvent nous permettre d'automatiser ces preuves.

1. L'algèbre des expressions fonctionnelles

1.1. Définition

a) les symboles fonctionnels

première partie

Soit FUNC l'ensemble des symboles

$\{ _0, _ \rightarrow _ ;, _ [], _ [], _ [], \dots, _ [...], \text{ALL}, _ *, \text{WH} \}$

FUNC n'est pas fini à cause de l'arité multiple de " $_ [...]$ ".

Soit PRIM l'ensemble des symboles

$\{ \text{id}, +, -, \text{mult}, \text{div}, \text{mod}, /, ?1, ?2, \dots$

$\text{last}, \text{cons}, \text{tl}, \text{trans}, \text{distl}, \text{append}, \text{iota} \}$

PRIM n'est pas fini à cause des sélecteurs.

Appelons CONST l'ensemble des notations des fonctions constantes obtenues par l'application de la forme fonctionnelle " $_$ " aux atomes de A.

$4^-, -5^-, 3.4^-,$

$\text{maison}^-, !^-, \text{t}^-$ et f^-

sont des exemples d'éléments de CONST.

PRIM $\cup$ CONST est l'ensemble des symboles fonctionnels d'arité 0 pour l'algèbre. C'est donc l'ensemble des symboles de constantes de l'algèbre.

L'ensemble des symboles fonctionnels de l'algèbre est alors

$$S = \text{FUNC} \cup \text{PRIM} \cup \text{CONST}$$

b) Une algèbre libre des expressions fonctionnelles

Appelons F le sous-ensemble des fonctions de profil $0 \rightarrow 0$ qui sont équivalentes aux programmes itératifs. F constitue le domaine de

première partie

l'algèbre. Un ensemble V de variables de fonctions est inclus dans F . Considérons les opérations OP dans F qui sont indexées par les symboles de S et qui possèdent la définition correspondante donnée au chapitre 1, paragraphe 2. L'algèbre des expressions fonctionnelles que nous appelons l'algèbre des FP-termes est la S -algèbre $\langle F, OP \rangle$. Signalons le point de vue de M. NIVAT ([Niv75], [Cou80]) qui considère la partie syntaxique des programmes comme élément d'un magma libre qui est analogue à notre S -algèbre.

1.2. Les FP-termes

Les éléments de V sont des variables de fonctions. Les éléments de la S -algèbre libre sur V sont appelés des termes (du premier ordre) que nous appellerons des FP-termes. Nous rappelons ici des concepts habituels pour les termes, dont G. HUET a donné des définitions formelles dans [Hue80a].

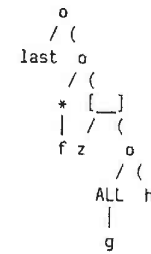
a) Représentation d'un terme par un arbre

Un terme est souvent représenté par un arbre dont les noeuds sont étiquetés par des symboles de S ou des variables de V .

Par exemple l'expression fonctionnelle

last o (*f o [z, ALL g o h])

est considérée comme un FP-terme. Elle peut être représentée par l'arbre suivant



b) Domaine d'un arbre et occurrences

Le domaine d'un arbre t , noté $D(t)$, est aussi appelé l'ensemble de ses occurrences. Les occurrences sont des mots du monoïde libre sur les entiers naturels positifs. Graphiquement, les occurrences sont les noeuds de l'arbre. La racine de l'arbre est désignée par l'occurrence vide que l'on peut noter " ϵ ". Le domaine de l'arbre en exemple est

$\{\epsilon, 1, 2, 21, 211, 22, 221, 222, 2221, 22211, 2222\}$

Le nombre d'occurrences d'un arbre t , qui est son nombre de noeuds, sera noté " $\#(t)$ ". Le nombre d'occurrences de l'arbre en exemple est 11.

Un arbre t est donc une application des occurrences dans l'ensemble S des étiquettes situées aux divers noeuds de l'arbre représentant un FP-terme. Pour l'exemple, nous avons

$t(\epsilon) = o, t(1) = \text{last}, t(2) = o, t(21) = *, t(211) = f, t(22) = [_],$

$t(221) = \forall, t(222) = o, t(2221) = \text{ALL}, t(22211) = g, t(2222) = h.$

"=" est ici le symbole pour "est défini par".

L'ensemble des occurrences est partiellement ordonné par l'ordre préfixe

qu'on note " ϵ ".

$u < v$ si et seulement si il existe w telle que $v = uw$

c) Sous-termes

On repère un sous-terme dans un terme par l'occurrence de sa racine dans le terme. Dans l'exemple,

le sous-terme " $*f$ " est situé à l'occurrence 21, il est aussi noté $t/21$,

" $ALL\ g$ " est situé à l'occurrence 2221 aussi noté $t/2221$,

" $[z, ALL\ g\ o\ h]$ " est situé à l'occurrence 22 aussi noté $t/22$.

Le terme complet est situé à l'occurrence vide " ϵ ".

Nous parlerons indifféremment de termes ou d'arbres, de sous-termes ou de sous-arbres, d'occurrences, de domaine d'arbre ou de noeuds. Nous désignerons l'ensemble des termes ou des arbres étiquetés (par les symboles S et par les variables V) par $T(S, V)$. Nous ne ferons pas de différence entre $T(S, V)$ et l'ensemble des FP-termes.

1.3. Opérations sur les termes : remplacement et substitution

a) Remplacement d'un sous-terme par un autre.

Etant donnés deux termes t , t' et une occurrence m de t , on peut remplacer dans t le sous-terme t/m par t' . Le résultat de cette opération est un terme t'' noté

$$t'' = t[m \leftarrow t'].$$

première partie

qui est tel que $t''(u) = t(u)$ si u n'a pas m comme préfixe

et $t''(mv) = t'(v)$.

Si, dans le terme de l'exemple, qu'on peut appeler t

$$t = \text{last } o (*f\ o\ [z, ALL\ g\ o\ h])$$

à l'occurrence 222, on remplace $t/222$ par le terme

$$t' = \text{trans } o\ [a, b],$$

on obtient un nouveau terme t''

$$t'' = t[222 \leftarrow (\text{trans } o\ [a, b])] = \text{last } o (*f\ o\ [z, \text{trans } o\ [a, b]])$$

b) ensemble des variables d'un terme

C'est l'ensemble des variables ayant au moins une occurrence dans le terme t . On le désigne par $V(t)$. Pour l'exemple

$$t'' = o \left(\begin{array}{c} / \left(\begin{array}{c} * \quad o \\ | \quad / \left(\begin{array}{c} f\ z \quad o \\ \quad \quad / \left(\begin{array}{c} \text{trans } [_] \\ \quad \quad / \left(\begin{array}{c} a \quad b \end{array} \end{array} \end{array} \end{array} \end{array} \end{array} \right. \right. \right. \end{array} \right. \right. \end{array} \right.$$

on a $V(t'') = \{f, z, a, b\}$.

On désigne par $O(t)$ l'ensemble des occurrences des symboles de S , c'est à dire des occurrences de non-variables. Pour l'exemple

$$O(t'') = \{\epsilon, 1, 2, 22, 221, 222\}.$$

première partie

c) opération de substitution

Une opération de substitution σ est une application de V dans les termes $T(S,V)$ telle que $\sigma(x) = x$ "presque partout", c'est-à-dire sauf sur un sous-ensemble fini de V appelé domaine de σ et noté $D(\sigma)$.

$D(\sigma)$ est égal aux x de V tels que $\sigma(x)$ soit différent de x .

Une substitution est représentée par son graphe, c'est-à-dire par un ensemble de couples notés sous la forme suivante

$$\{(x \rightarrow \sigma(x)) \text{ pour les } x \text{ de } D(\sigma)\}$$

Pour le terme

$$t'' = \text{last } o (*f \ o \ [z, \text{trans } o \ [a, b]])$$

on peut définir une substitution

$\sigma_1 =$

$$\{(f \rightarrow \text{mult}), (z \rightarrow 1^-), (a \rightarrow \text{ALL } a \ o \ (\text{iota } o \ n)), (b \rightarrow \text{ALL } b \ o \ (\text{iota } o \ n))\}$$

On obtient le terme

$$\sigma_1(t'') = \text{last } o (*\text{mult } o$$

$$[1^-, \text{trans } o \ [\text{ALL } a \ o \ (\text{iota } o \ n), \text{ALL } b \ o \ (\text{iota } o \ n)]]).$$

La substitution σ_1 fait apparaître une nouvelle variable n . On a

$$V(\sigma_1(t'')) = \{a, b, n\}$$

Il est possible de composer entre elles des substitutions.

Considérons par exemple une substitution σ_2

$$\sigma_2 = \{(n \rightarrow 5^-), (a \rightarrow 2^-)\}$$

première partie

On peut définir une substitution σ_3 composée des substitutions σ_1 et σ_2 et notée $\sigma_2 \sigma_1$. On a alors

$$\sigma_2(\sigma_1(t'')) =$$

$$\text{last } o (*\text{mult } o \ [1^-, \text{trans } o \ [\text{ALL } 2^- \ o \ (\text{iota } o \ 5^-), \\ \text{ALL } b \ o \ (\text{iota } o \ 5^-)]])$$

$$\sigma_3 = \sigma_2 \sigma_1 =$$

$$\{(f \rightarrow \text{mult}), (z \rightarrow 1^-), (a \rightarrow \text{ALL } 2^- \ o \ (\text{iota } o \ 5^-)) \\ , (b \rightarrow \text{ALL } b \ o \ (\text{iota } o \ 5^-))\}$$

d) Les termes clos

La S -algèbre libre sur un ensemble de variables vide est l'algèbre initiale. Cette algèbre initiale existe car l'ensemble des constantes ($\text{PRIM} \cup \text{CONST}$) n'est pas vide.

Ses termes, qui ne contiennent pas de variables sont appelés des termes clos.

Si nous appliquons au terme t

$$t = \text{last } o (*f \ o \ [z, \text{ALL } g \ o \ h])$$

la substitution σ

$$\sigma = \{(f \rightarrow +), (z \rightarrow 0^-), (g \rightarrow \text{mult}), (h \rightarrow \text{trans})\},$$

nous obtenons le terme clos

$$\sigma(t) = \text{last } o (*+ \ o \ [0^-, \text{ALL } \text{mult } o \ \text{trans}])$$

Nous allons maintenant définir des axiomes qui sont des équations entre FP-termes. Ces axiomes serviront à prouver d'autres équations valides dans

première partie

la FP-algèbre.

2. Equations valides dans la FP-algèbre

2.1. Les relations d'équivalences

a) égalité extensionnelle

Pour justifier nos transformations, il faut que deux termes soient équivalents si et seulement si ils définissent la même fonction dans la FP-algèbre.

L'équivalence est donc l'égalité extensionnelle que nous noterons

$=_{\text{ext}}$

Nous la définissons par

$f =_{\text{ext}} g$ si et seulement si pour tout objet x , $f : x == g : x$

~~~~~

Nous avons par exemple

$+ \circ [4^-, 3^-] =_{\text{ext}} 7^-$

mais  $"/ \circ [\text{id}, \text{id}]$  et  $"1^-$  ne sont pas extensionnellement égaux.

En effet

$(/ \circ [\text{id}, \text{id}]) : 0 == / : [0, 0] == !$

alors que  $1^- : 0 == 1$

première partie

#### b) égalité sous condition

Nous aimerions pouvoir exprimer aussi le fait que deux fonctions  $f$  et  $g$  ne sont équivalentes que sur un certain domaine  $D$  strictement inclus dans l'ensemble des objets  $O$ .

Nous appelons prédicats les fonctions qui peuvent avoir pour résultat "T". Nous dirons qu'un prédicat  $p$  cerne un domaine  $D$  si et seulement si

a) pour tout objet  $x$  de  $D$ , on a  $p : x == T$

b) pour tout objet  $x$  qui n'est pas dans  $D$ , on a  $p : x == F$  ou  $p : x == !$

En utilisant un prédicat  $p$  qui cerne un domaine  $D$ , nous pouvons dire que deux fonctions  $f$  et  $g$  sont les mêmes sur le domaine  $D$  si et seulement si pour tout  $x$  tel que  $p : x == T$ , on a  $f : x == g : x$ .

Nous définissons ainsi une égalité conditionnelle:

$p :: f = g$   
~~~~~

si et seulement si pour tout objet x , $p : x == T \Rightarrow f : x == g : x$
~~~~~

( $\Rightarrow$  est mis pour "implique").

Un exemple peut être

$T^- \circ (/ \circ [x, x]) :: / \circ [x, x] = 1^-$  (ici  $x$  est une fonction).

Pour les objets  $z$  tels que

$/ : [x : z, x : z] == !$ , on a

$T^- \circ (/ \circ [x, x]) : z == T^- : (/ : [x : z, x : z])$   
 $== T^- : ! == !$ .

première partie

Dans le cas contraire, soit  $y$  l'objet résultat de la division.  $y$  est différent de "!" et

$$T^- : y == T.$$

On a alors:  $/ o [x:z, x:z] == y == 1$

Remarquons que si  $f$  et  $g$  sont extensionnellement égales, on peut toujours écrire

$$T^- :: f = g$$

## 2.2. Equations valides dans la FP-algèbre

### a) Définitions

Une équation est une paire de termes  $t_1$  et  $t_2$  que l'on note habituellement  $t_1 = t_2$ .

$t_1$  et  $t_2$  sont parfois appelés respectivement premier et second membre de l'équation.

Pour la FP-algèbre, une équation  $t_1 = t_2$  n'est valide que si  $t_1$  et  $t_2$  définissent la même fonction.

L'égalité extensionnelle nous permet alors de prouver qu'une équation est valide pour la FP-algèbre:

Une équation  $t_1 = t_2$  est valide dans la FP-algèbre

~~~~~

si et seulement si $t_1 =_{\text{ext}} t_2$.

~~~~~

Par exemple l'équation

$$(f \circ g) \circ h = f \circ (g \circ h)$$

représente la loi d'associativité de la forme fonctionnelle "o". Pour prouver que cette équation est valide dans la FP-algèbre, il suffit de montrer que pour tout objet  $x$ , on a

$$((f \circ g) \circ h) : x == (f \circ (g \circ h)) : x.$$

Il faut appliquer la définition de la forme fonctionnelle "o" donnée au chapitre I paragraphe 2.5. Alors, pour tout objet  $x$

$$((f \circ g) \circ h) : x == (f \circ g) : (h : x)$$

$$== f : (g : (h : x))$$

$$\text{et } (f \circ (g \circ h)) : x == f : ((g \circ h) : x)$$

$$== f : (g : (h : x)).$$

J. BACKUS [Bac78] cite et prouve ainsi un ensemble d'équations valides dans l'algèbre des expressions fonctionnelles. Nous allons donner trois exemples d'équations et de leurs preuves en utilisant l'égalité extensionnelle.

### b) Exemples de preuves de la validité d'équations dans la FP-algèbre

Le premier exemple représente une loi sur les deux primitives "last" et "dist1". Le deuxième exemple est une loi de composition de deux fonctions ALL  $f$  et ALL  $g$ . Le troisième exemple concerne une loi sur la forme fonctionnelle "\*" qui correspond à la propriété de "fusion" de deux "itérations séparées" en une seule.

#### PREMIER EXEMPLE

Preuve de la validité dans la FP-algèbre de l'équation

LDISTL:  $\text{last } o \text{ (distl } o [x,a]) = [x, \text{last } o a]$   
 ~~~~~

Nous utilisons les définitions des formes fonctionnelles "o" et "[_]" données au chapitre I paragraphe 2.5 et les définitions des primitives "last" et "distl" données au chapitre I paragraphe 2.4.

Pour tout objet z

$(\text{last } o \text{ (distl } o [x,a])) : z == \text{last} : ((\text{distl } o [x,a]) : z)$
 $== \text{last} : (\text{distl} : ([x,a] : z))$
 $== \text{last} : (\text{distl} : [x:z, a:z])$
 $[x, \text{last } o a] : z == [x:z, (\text{last } o a):z]$
 $== [x:z, \text{last} : (a:z)].$

Nous distinguons trois cas

a) a:z n'est pas une séquence

$(\text{last } o \text{ (distl } o [x,a])) : z == \text{last} : (\text{distl} : [x:z, a:z])$
 $== \text{last} : ! == !$
 $[x, \text{last } o a] : z == [x:z, \text{last} : (a:z)] == [x:z, !] == !$

b) a:z est une séquence vide: $a:z == \langle \rangle$

$(\text{last } o \text{ (distl } o [x,a])) : z == \text{last} : (\text{distl} : [x:z, a:z])$
 $== \text{last} : (\text{distl} : [x:z, \langle \rangle])$
 $== \text{last} : \langle \rangle$
 $== !$
 $[x, \text{last } o a] : z == [x:z, \text{last} : (a:z)]$
 $== [x:z, \text{last} : \langle \rangle]$
 $== [x:z, !]$
 $== !$

première partie

c) a:z est une séquence non vide: $a:z == \langle z_1, z_2, \dots, z_n \rangle$

$(\text{last } o \text{ (distl } o [x,a])) : z == \text{last} : (\text{distl} : [x:z, a:z])$
 $== \text{last} : (\text{distl} : [x:z, \langle z_1, z_2, \dots, z_n \rangle])$
 $== \text{last} : \langle [x:z, z_1], [x:z, z_2], \dots, [x:z, z_n] \rangle$
 $== [x:z, z_n]$
 $[x, \text{last } o a] : z == [x:z, \text{last} : (a:z)]$
 $== [x:z, \text{last} : \langle z_1, z_2, \dots, z_n \rangle]$
 $== [x:z, z_n]$

DEUXIEME EXEMPLE

Preuve de la validité dans la FP-algèbre de l'équation

ALLALL: $\text{ALL } f \text{ } o \text{ ALL } g = \text{ALL } (f \text{ } o \text{ } g)$
 ~~~~~

Nous utilisons la définition de "o" donnée au chapitre I paragraphe 2.5.

Pour tout objet x,

$(\text{ALL } f \text{ } o \text{ ALL } g) : x == (\text{ALL } f) : ((\text{ALL } g) : x)$

Nous utilisons la définition de "ALL f" donnée au chapitre I paragraphe 2.6. Nous distinguons trois cas

a) x n'est pas une séquence

$(\text{ALL } f \text{ } o \text{ ALL } g) : x == (\text{ALL } f) : ((\text{ALL } g) : x) == (\text{ALL } f) : ! == !$   
 $(\text{ALL } (f \text{ } o \text{ } g) : x) == !$

b) x est une séquence vide:  $x == \langle \rangle$

$(\text{ALL } f \text{ } o \text{ ALL } g) : x == (\text{ALL } f) : ((\text{ALL } g) : x) == (\text{ALL } f) : ((\text{ALL } g) : \langle \rangle)$   
 $== (\text{ALL } f) : \langle \rangle == \langle \rangle$

$(\text{ALL } (f \text{ } o \text{ } g)) : x == (\text{ALL } (f \text{ } o \text{ } g)) : \langle \rangle == \langle \rangle$

c) x est une séquence non vide:  $x == \langle x_1, x_2, \dots, x_n \rangle$

première partie

$$\begin{aligned}
 (\text{ALL } f \circ \text{ALL } g) : x &== (\text{ALL } f) : ((\text{ALL } g) : x) \\
 &== (\text{ALL } f) : ((\text{ALL } g) : \langle x_1, x_2, \dots, x_n \rangle) \\
 &== (\text{ALL } f) : \langle g : x_1, g : x_2, \dots, g : x_n \rangle \\
 &== \langle f : (g : x_1), f : (g : x_2), \dots, f : (g : x_n) \rangle \\
 (\text{ALL } (f \circ g)) : x &== \text{ALL } (f \circ g) : \langle x_1, x_2, \dots, x_n \rangle \\
 &== \langle (f \circ g) : x_1, (f \circ g) : x_2, \dots, (f \circ g) : x_n \rangle \\
 &== \langle f : (g : x_1), f : (g : x_2), \dots, f : (g : x_n) \rangle
 \end{aligned}$$

## TROISIEME EXEMPLE

Preuve de la validité dans la FP-algèbre de l'équation

F\*:  $\text{trans } o [ *f \circ [x,a] , *g \circ [y,a] ]$

$= * [ f \circ [?1o?1,?2] , g \circ [?2o?1,?2] ] \circ [[x,y],a]$

Regardons ce que signifie cette équation pour les programmes itératifs.

Les deux itérations "séparées" " $*f$ " et " $*g$ " s'appliquent à la même séquence (résultat de la fonction  $a$ ). Si " $*f$ " a pour résultat la séquence " $\langle x_1, x_2, \dots, x_n \rangle$ " et " $*g$ " la séquence " $\langle y_1, y_2, \dots, y_n \rangle$ ", le résultat du membre gauche est la séquence des couples " $\langle [x_1, y_1], [x_2, y_2], \dots, [x_n, y_n] \rangle$ ". Le membre droit donne le même résultat avec une seule itération " $*$ ". Cette propriété porte souvent le nom de "fusion d'itérations".

Pour la preuve, nous allons utiliser les définitions de " $o$ ", " $[_]$ ", " $*$ ", " $\text{trans}$ ", " $?1$ " et " $?2$ " données au chapitre I paragraphe 2. La définition de " $*$ " du chapitre I paragraphe 2.6. est récurrente sur la structure de la séquence deuxième élément du couple argument d'une fonction " $*f$ ". Nous effectuons donc un raisonnement par récurrence sur la structure de cette séquence.

première partie

Pour tout objet  $z$ ,

$$\begin{aligned}
 (\text{trans } o [ *f \circ [x,a] , *g \circ [y,a] ]) : z &== \text{trans} : ([ *f \circ [x,a] , *g \circ [y,a] ]) : z \\
 &== \text{trans} : [ (*f \circ [x,a]) : z , (*g \circ [y,a]) : z ] \\
 &== \text{trans} : [ *f : [x : z, a : z] , *g : [y : z, a : z] ] \quad (1) \\
 (*f \circ [?1o?1,?2] , g \circ [?2o?1,?2]) \circ [[x,y],a] : z \\
 &== (*f \circ [?1o?1,?2] , g \circ [?2o?1,?2]) : ([ [x,y],a ] : z) \\
 &== (*f \circ [?1o?1,?2] , g \circ [?2o?1,?2]) : [ [x : z, y : z] , a : z ] \quad (2)
 \end{aligned}$$

Il suffit de montrer que  $(1) == (2)$ .

Nous distinguons trois cas:

a)  $a : z$  n'est pas une séquence, alors

$*f : [x : z, a : z] == !$

$*g : [y : z, a : z] == !$

$(1) == \text{trans} : [!,!] == \text{trans} : ! == !$

et

$(2) == (*f \circ [?1o?1,?2] , g \circ [?2o?1,?2]) : [ [x : z, y : z] , a : z ] == !$

b)  $a : z$  est une séquence

Nous raisonnons par récurrence. Le cas de base est celui de la séquence vide de longueur 0. Toute séquence  $s$  non vide s'écrit  $\text{cons} : \langle s_1, u \rangle$ . Si  $s$  est de longueur  $n$ , la longueur de  $u$  est  $n-1$ . Le raisonnement par récurrence sur la structure d'une séquence est un corollaire de la récurrence sur les entiers. Pour prouver une propriété  $P(s)$  pour toute séquence  $s$ , il suffit de prouver

1)  $P(\langle \rangle)$  cas de base

2)  $P(\text{cons} : [s_1, u])$  avec l'hypothèse de récurrence  $P(u)$ .

1)  $a : z == \langle \rangle$

première partie

$*f:[x:z,a:z] == *f:[x:z,<>] == <x:z>$   
 $*g:[y:z,a:z] == *g:[y:z,<>] == <y:z>$   
 $(1) == \text{trans}:[<x:z>,<y:z>] == <[x:z,y:z]>$   
 et  
 $(2) == (*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):[[x:z,y:z],a:z]$   
 $==(*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):[[x:z,y:z],<>] == <[x:z,y:z]>$   
 2)  $a:z == \text{cons}:[s_1,u]$

L'hypothèse de récurrence P(u) donne pour tout objet x' et y'

$\text{trans}:[*f:[x',u], *g:[y',u]] == (*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):[[x',y'],u]$   
 ~~~~~~

LEMME

Pour tout objet x' et y', pour deux séquences quelconques u et v de même longueur

$\text{trans}:[\text{cons} \text{ o } [x',u], \text{cons} \text{ o } [y',v]] == \text{cons} \text{ o } [[x',y'], \text{trans}:[u,v]].$

La démonstration de "(1) == (2)" est simple par récurrence sur la structure des séquences de même longueur u et v

$*f:[x:z, \text{cons}:[s_1,u]] == \text{cons}:[x:z, *f:[f:[x:z,s_1],u]]$
 $*g:[y:z, \text{cons}:[s_1,u]] == \text{cons}:[y:z, *g:[g:[y:z,s_1],u]]$
 $(1) == \text{trans}:[\text{cons} \text{ o } [x:z, *f:[f:[x:z,s_1],u]],$
 $\text{cons} \text{ o } [y:z, *g:[g:[y:z,s_1],u]]]$

En appliquant le lemme, on a donc

$(1) == \text{cons}:[x:z,y:z, \text{trans}:[*f:[f:[x:z,s_1],u], *g:[g:[y:z,s_1],u]]]$
 $(2) == (*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):[[x:z,y:z], \text{cons}:[s_1,u]]$
 $== \text{cons} \text{ o } [[x:z,y:z], (*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):t]$
 avec $t == [[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]:[[x:z,y:z],s_1],u]$

première partie

$t == [(f \text{ o } [?1o?1,?2]):[[x:z,y:z],s_1],$
 $(g \text{ o } [?2o?1,?2]):[[x:z,y:z],s_1]],u]$
 $== [[f:([?1o?1,?2]):[[x:z,y:z],s_1]],u]$
 $g:([?2o?1,?2]):[[x:z,y:z],s_1]],u]$
 $== [[f:([?1o?1]):[[x:z,y:z],s_1],?2:([x:z,y:z],s_1)],u]$
 $g:([?2o?1]):[[x:z,y:z],s_1],?2:([x:z,y:z],s_1)],u]$
 $== [[f:[?1:([?1:([x:z,y:z],s_1)],s_1)],u]$
 $g:[?2:([?1:([x:z,y:z],s_1)],s_1)],u]$
 $== [[f:[?1:[x:z,y:z],s_1],$
 $g:[?2:[x:z,y:z],s_1]],u]$
 $== [[f:[x:z,s_1],g:[y:z,s_1]],u]$

En reportant t dans (2), on a

$(2) == \text{cons}:[x:z,y:z, (*[f \text{ o } [?1o?1,?2], g \text{ o } [?2o?1,?2]]):$
 $[[f:[x:z,s_1],g:[y:z,s_1]],u]]$

On a trouvé que

$(1) == \text{cons}:[x:z,y:z, \text{trans}:[*f:[f:[x:z,s_1],u], *g:[g:[y:z,s_1],v]]]$

On peut maintenant appliquer l'hypothèse de récurrence avec

$x' == f:[x:z],s_1]$

$y' == g:[y:z],s_2]$

pour avoir (1) == (2).

EN CONCLUSION

La preuve de validité d'une équation quelconque dans la FP-algèbre est aussi simple que pour les exemples que nous venons de donner. Elle pourrait même être automatisée (en utilisant par exemple le système LCF [Gor77] dont l'objectif initial était de prouver des équations de ce type).

première partie

Ultérieurement, nous ne détaillerons pas les preuves des équations par l'égalité extensionnelle: elles se ressemblent toutes.

2.3. Equations conditionnelles valides dans la FP-algèbre

a) Définitions

Une équation conditionnelle entre deux termes t_1 et t_2 est de la forme

$$p :: t_1 = t_2$$

p est un terme particulier qui s'interprète comme un prédicat et que l'on appelle "précondition".

Pour la FP-algèbre, une équation conditionnelle n'est valide que si, sur le domaine cerné par le prédicat p , t_1 et t_2 définissent la même fonction. L'égalité conditionnelle permet alors de prouver qu'une équation conditionnelle est valide pour la FP-algèbre.

Une équation conditionnelle $p :: t_1 = t_2$ est valide

si et seulement si pour tout x $p:x \Rightarrow t_1:x == t_2:x$

L'équation conditionnelle

$$T^- \circ (/ \circ [x,x]) :: / \circ [x,x] = I^-$$

est valide dans la FP-algèbre.

NOTATION

première partie

Nous noterons la precondition T^- par DEF.

"DEF o g"

a (comme " T^- o g") la signification:

"là où g est définie".

REMARQUE

Une équation peut toujours se mettre sous une forme conditionnelle car

$$f = g \text{ si et seulement si } T^- :: f = g$$

b) Exemples

$$1) P3: (DEF \circ g) :: \exists ! x [f, g] = f$$

En effet, pour tout objet x ,

$$(DEF \circ g) : x == DEF : (g : x).$$

Dans le cas où $g:x == !$, c'est-à-dire pour un objet x hors du domaine de définition de g ,

$$(DEF \circ g) : x == DEF : ! == !.$$

Le prédicat ne vaut pas "T".

Dans le cas contraire, c'est-à-dire pour un objet x du domaine de définition de g

$$(DEF \circ g) : x == T$$

Le prédicat vaut "T".

première partie

En ce cas, on a

```
(?1 o [f,g]) : x == ?1 : ([f,g] : x)
      == ?1 : [f:x,g:x]
      == f:x
```

2) LTRANS: DEF o (trans o [a,b])
 ~~~~~

```
:: last o (trans o [a,b]) = [last o a, last o b]
~~~~~
```

Pour tout objet x,

```
p:x == (DEF o (trans o [a,b])) : x
 == DEF : (trans : [a:x,b:x])
```

Dans le cas où "a:x" et "b:x" ont pour résultat deux séquences u et v de longueur égales, "trans : [a:x,b:x]" ne vaut pas "!".

Dans les autres cas, "p:x" vaut "!". Si "a:x" et "b:x" sont deux séquencés de même longueur, "p:x" vaut "T".

On a alors:

```
(last o (trans o [a,b])):x == last : (trans : [a:x,b:x])
```

```
[last o a, last o b]:x == [last : (a:x), last : (b:x)]
```

Posons a:x == <a1,a2,...,an>

et b:x == <b1,b2,...,bn>, on trouve

```
(last o (trans o [a,b]))
```

```
== last : <[a1,b1],[a2,b2],...,[an,bn]>
```

```
== [an,bn]
```

```
[last o a, last o b] : x
```

```
== [last:<a1,a2,...,an>, last:<b1,b2,...,bn>]
```

```
== [an,bn]
```

première partie

## 2.4. Algèbre et théorie équationnelle

### a) Définition

Etant donné un ensemble d'équations A, on peut considérer la classe des algèbres qui valident toutes les équations de A. Cette classe d'algèbre s'appelle alors la classe des modèles de A que l'on note M(A). En particulier, si nous retenons un ensemble d'équations A valides dans la FP-algèbre, la FP-algèbre est un modèle de A.

Etant donnée une équation, on veut alors savoir si elle est valide (donc en particulier pour la FP-algèbre). C'est ce qu'on appelle le problème de la A-validité que l'on peut noter

$$A \models t_1 = t_2$$

La plus petite congruence de l'algèbre libre  $T(S,V)$  contenant toutes les paires  $\{\sigma(t_1), \sigma(t_2)\}$ , où " $t_1=t_2$ " est une équation de A et  $\sigma$  une substitution, existe et est appelée A-égalité. On la note  $=_A$ . Deux FP-termes  $t_1$  et  $t_2$  tels que

$$t_1 =_A t_2, \text{ sont dits A-égaux.}$$

La théorie équationnelle engendrée par A est l'algèbre quotient

$$(T(S,V)/=_A).$$

Le théorème de la complétude [Bir35] (dû à BIRKHOFF) permet d'étudier la validité dans M(A) de manière purement syntaxique par l'intermédiaire de la A-égalité. En effet, BIRKHOFF prouve que

première partie

$A \models t_1 = t_2$  si et seulement si  $t_1 \approx_A t_2$

~~~~~

Ainsi si deux expressions fonctionnelles sont interprétées comme deux FP-termes  $t_1$  et  $t_2$  et que  $t_1$  et  $t_2$  sont A-égaux, alors " $t_1 = t_2$ " est une équation de la FP-algèbre,  $t_1$  et  $t_2$  sont extensionnellement égaux, les deux expressions fonctionnelles sont équivalentes et elles représentent la même fonction.

$t_1 \approx_A t_2 \Rightarrow t_1 = t_2$

~~~~~

Nous voulons aussi tenir compte des équations conditionnelles valides dans la FP-algèbre.

#### b) Théorie équationnelle intégrant des équations conditionnelles

Etant donné un ensemble d'équations conditionnelles A (A peut contenir des équations, il suffit de leur attribuer la précondition " $\top$ "), on peut considérer la classe des algèbres qui valident toutes les équations conditionnelles de A. C'est la classe des modèles de A que l'on note  $M(A)$ . En particulier, si nous retenons un ensemble d'équations conditionnelles A valides dans la FP-algèbre, la FP-algèbre est un modèle de A.

On se pose aussi le problème de la A-validité d'une équation conditionnelle.

On peut considérer qu'une congruence que l'on peut noter " $\approx$ " valide A si et seulement si, pour toute équation " $p :: t_1 = t_2$ " de A et pour toute substitution  $\sigma$ , on a

première partie

$\sigma(p) \leq \top \Rightarrow \sigma(t_1) \leq \sigma(t_2)$ .

L'égalité extensionnelle valide A:

si  $\sigma(p) =_{\text{ext}} \top$  alors  $\sigma(p):x = \top$  pour tout x,

ce qui implique  $\sigma(t_1):x = \sigma(t_2):x$  pour tout x et donc:

$\sigma(t_1) =_{\text{ext}} \sigma(t_2)$ .

J. L. REMY [Rem82] montre l'existence d'une plus petite congruence sur  $T(S,V)$  parmi les congruences qui valident A qu'il appelle la A-égalité, notée aussi  $\approx_A$ .

~~~~~

Deux termes  $t_1$  et  $t_2$  tels que

$t_1 \approx_A t_2$  sont dits A-égaux.

La théorie conditionnelle engendrée par A est l'algèbre quotient  $(T(S,V)/\approx_A)$ .

La A-égalité valide A:

Pour toute substitution  $\sigma$  et pour toute équation

$p :: t_1 = t_2$  de A,

$\sigma(p) \approx_A \top \Rightarrow \sigma(t_1) \approx_A \sigma(t_2)$ .

Nous avons alors un théorème de complétude: l'équation conditionnelle

$A \models p :: t_1 = t_2$

si et seulement si  $p \approx_A \top \Rightarrow t_1 \approx_A t_2$

première partie

Ainsi, si deux expressions fonctionnelles sont considérées comme deux FP-termes  $t_1$  et  $t_2$  et que  $t_1$  et  $t_2$  sont A-égaux, alors  $t_1 = t_2$  est une équation de la FP-algèbre,  $t_1$  et  $t_2$  sont extensionnellement égaux, les deux expressions fonctionnelles sont équivalentes et elles représentent la même fonction.

La A-égalité est incluse dans l'égalité extensionnelle. Nous pouvons donc prouver l'équivalence en utilisant la A-égalité.

L'utilisation de l'égalité extensionnelle (et par conséquent de la A-égalité) réduit considérablement le champ d'application des équations.

#### EXEMPLES

Pour l'équation

P3: DEF o g :: ?1 o [f,g] = f, on a

$$\sigma(?1 \text{ o } [f,g]) =_{\text{ext}} \sigma(f),$$

pour toute substitution  $\sigma$  telle que

$$\sigma(\text{DEF o g}) =_{\text{ext}} \text{I}^-.$$

Donc  $\sigma(g)$  ne doit pas avoir pour résultat "!". Autrement dit le domaine de définition de  $g$  doit être 0. On dit alors que  $g$  est "partout définie" ou encore "totale". Cependant, sur le domaine de définition de  $\sigma(g)$ , les termes  $\sigma(?1 \text{ o } [f,g])$  et  $\sigma(f)$  sont équivalents.

Pour l'équation

LTRANS: DEF o (trans o [a,b])

$$:: \text{last o (trans o [a,b])} = [\text{last o a}, \text{last o b}], \text{ on a}$$

$$\sigma(\text{last o (trans o [a,b])}) =_{\text{ext}} \sigma([\text{last o a}, \text{last o b}])$$

pour toute substitution  $\sigma$  telle que

$$\sigma(\text{trans o [a,b]}) =_{\text{ext}} \text{I}^-$$

Autrement dit,  $\sigma(a)$  et  $\sigma(b)$  doivent avoir "partout" pour résultat des séquences de même longueur.

Mais on peut aussi dire que ces termes sont équivalents là où  $\sigma(a)$  et  $\sigma(b)$  ont pour résultat des séquences de même longueur c'est-à-dire quand  $\sigma(\text{trans o [a,b]})$  est défini.

Pour l'équation

$$\text{DEF o } (/ \text{ o } [x, x]) :: / \text{ o } [x, x] = \text{I}^-,$$

les termes  $\sigma(/ \text{ o } [x, x])$  sont équivalents à  $\text{I}^-$  pour toute substitution  $\sigma$  telle que  $\sigma(\text{DEF o } (/ \text{ o } [x, x]))$  soit équivalente à  $\text{I}^-$ .  $\sigma(x)$  ne peut donc avoir un résultat nul et  $x$  doit être partout définie. Mais on peut cependant dire que ces termes sont équivalents là où  $x$  est définie et où  $\sigma(x)$  n'a pas un résultat nul.

L'exemple le plus frappant est peut être celui de l'équation

$$\text{DEF o (last o tl)} :: \text{last o tl} = \text{last}$$

laquelle "DEF o (last o tl)" ne peut être extensionnellement égale " $\text{I}^-$ " puisque "last" n'est pas partout définie. Cet axiome signifie que "last o tl" a un domaine de définition inclus dans "last" et que le remplacement de "last" par "last o tl" ne peut se faire que là où "last o tl" est définie.

Remarquons que nos axiomes conditionnels ont une précondition de la forme "DEF o p", ce sont des équations conditionnelles particulières.

Pour une équation conditionnelle "DEF o p ::  $t_1 = t_2$ ", nous allons montrer que nous n'aurons pas à prouver la précondition "DEF o p".

c) Comment évite-t-on de prouver la précondition avant d'appliquer une équation conditionnelle ?

Nous nous servons de l'hypothèse suivante: Les expressions e des fonctions f que nous considérons sont en principe "correctes" et donc il ne se produit pas d'erreur d'exécution pour les programmes qu'elles codent. Autrement dit il n'y a pas de fonctions composantes produisant "!".

Ce qui veut dire en particulier que tout sous-terme t de e n'aura jamais pour résultat "!" lors d'une application quelconque de f à un objet x de son domaine de définition. On a alors t défini sur son domaine d'application dans e et donc

$$(DEF \text{ o } t) : x = I$$

Nous disons que nous "appliquons l'équation"

$$p :: t_1 = t_2$$

à l'occurrence v dans l'expression e si il existe une substitution  $\sigma$  telle que

$$e/v = \sigma(t_1) \text{ (ou } e/v = \sigma(t_2))$$

on obtient l'expression

$$e'' =_A e$$

par remplacement du sous-terme e/v par le sous-terme égal  $\sigma(t_1)$  (ou  $\sigma(t_2)$ )

première partie

$$e'' = e[v \leftarrow \sigma(t_1)] \text{ (ou } e[v \leftarrow \sigma(t_2)])$$

Mais  $\sigma(t_1)$  n'est égal à  $\sigma(t_2)$  que si l'on a prouvé que  $\sigma(p)$  appliqué à tout objet du domaine de définition de e vaut "I".

Considérons maintenant une équation "DEF o p ::  $t_1 = t_2$ ". Elle traduit le plus généralement le fait que le domaine de définition de  $t_1$  est inclus dans celui de  $t_2$  (ou l'inverse), ce que l'on trouve souvent exprimé par la relation "est moins défini que" que l'on peut noter " $\subset$ ". Cette relation peut se traduire syntaxiquement dans la précondition. p est un sous-terme de  $t_1$  (ou  $t_2$ ) à l'occurrence u et u est telle que, si  $t_1$  (ou  $t_2$ ) s'applique à un objet x, " $t_1/u$ " (ou " $t_2/u$ ") s'applique aussi à l'objet x.

Or quand nous appliquons à une expression e "correcte" une équation conditionnelle "DEF o p ::  $t_1 = t_2$ " valide,  $\sigma(p) = \sigma(t_1/u)$  (ou  $\sigma(t_2/u)$ ) est sous terme de e et ne peut donc pas produire "!" pour les applications de e sur son domaine de définition.  $\sigma(DEF \text{ o } t)$  vaut "I", on n'a pas besoin de le prouver.

#### EXEMPLE

On montre que les expressions (2) et (3) de l'exemple 3 du chapitre I

$$(2) / o (\text{last } o (\text{trans } o [u, v]))$$

$$(3) / o [\text{last } o u, \text{last } o v]$$

sont A-égales en utilisant l'équation conditionnelle LTRANS citée ci-dessus.

LTRANS: DEF o (trans o [a, b]) ::

$$\text{last } o (\text{trans } o [a, b]) = [\text{last } o a, \text{last } o b]$$

Pour LTRANS, on a

première partie

$$\text{last } o \text{ (trans } o \text{ [a,b])} \subset [\text{last } o \text{ a, last } o \text{ b}]$$

L'hypothèse est que (2) est correcte donc "DEF o (trans o [u,v])" vaut "T".

On peut donc appliquer LTRANS en remplaçant "last o (trans o [u,v])" par "[last o u, last o v]" à l'occurrence 2 de l'expression (2) et on a

$$(2) =_A (3).$$

Nous pouvons remarquer que nous n'aurions pas pu partir de l'équation (3): Nous ne pouvons pas remplacer "[last o u, last o v]" par "last o (trans o [u,v])" sans prouver que "trans o [u,v]" est défini.

#### EN CONCLUSION

Pour une équation conditionnelle de la forme "DEF o p:  $t_1 = t_2$ " qui exprime que " $t_1 \text{ Ct } t_2$ " appliquée à une expression e correcte grâce à un filtrage de  $t_1$ , nous navons pas à prouver la précondition.

### 3. Axiomes d'une théorie équationnelle

Nous voulons constituer un ensemble d'équations A valides dans la FP-algèbre qui soient des axiomes pour la A-égalité.

Nous avons vu au paragraphe 2 que la A-égalité est alors incluse dans l'égalité extensionnelle.

Cependant il est impossible de trouver un ensemble fini (décidable) A d'axiomes tel que l'on ait la réciproque (en lambda calcul, l'égalité extensionnelle devient la  $\eta$ -conversion).

première partie

Mais nous voulons que A soit assez riche pour que l'on puisse prouver l'équivalence de deux expressions pour un ensemble suffisamment large d'expressions. Ce sont celles que l'on obtient pour coder un programme itératif quand on utilise un mode de construction structuré avec des séquences. Les expressions données comme exemples au chapitre I paragraphe 3 nous serviront encore ici d'exemples. Nous voulons que, si une classe de termes A-égaux contient un terme qui peut être trouvé par un raisonnement structuré, cette classe contienne aussi les termes trouvés par les autres possibilités de raisonnements structurés.

Nous avons rassemblé les axiomes de A dans l'annexe I. Dans ce paragraphe, nous introduisons la plupart des axiomes de A. Nous illustrons A en prouvant que les expressions trouvées pour les exemples du chapitre 1 sont A-égales.

#### 3.1. Axiomes de base

La plupart des axiomes que nous proposons sont cités par J. BACKUS.

#### La composition des fonctions est associative

$$E1: (f \circ g) \circ h = f \circ (g \circ h)$$

#### La composition des fonctions est distributive à gauche sur [...]

$$E2: [f1, f2, \dots, fn] \circ h = [f1 \circ h, f2 \circ h, \dots, fn \circ h]$$

E2 note une infinité d'axiomes : un axiome pour chaque arité de [...].

On a:

Première partie

E2<sub>1</sub>: [f1] o h = [f1 o h]  
 E2<sub>2</sub>: [f1,f2] o h = [f1 o h, f2 o h] ...

"\_" construit des fonctions absorbantes à droite pour la composition

E3: DEF o y :: x<sup>-</sup> o y = x<sup>-</sup>  
 ~~~~~

Cet axiome est conditionnel, on a "x<sup>-</sup> o y C x<sup>-</sup>".

L'identité est élément neutre pour la composition

P1: id o f = f  
 ~~~~~

P2: f o id = f  
 ~~~~~

Axiomes pour les sélecteurs

Pour i = 1,2,...,n

P3: DEF o [f1,f2,...,fn] :: ?i o [f1,f2,...,fn] = fi  
 ~~~~~

On a "?i o [f1,f2,...,fn] C fi"

P3 exprime une infinité d'axiomes:

- une infinité pour le sélecteur ?1:

P3<sub>11</sub>: ?1 o [f1] = f1

P3<sub>12</sub>: DEF o f2 :: ?1 o [f1,f2] = f1

P3<sub>13</sub>: DEF o [f1,f2,f3] :: ?1 o [f1,f2,f3] = f1

(pour chaque arité de [...]),

première partie

- une infinité pour le sélecteur ?2:

P3<sub>22</sub>: DEF o f1 :: ?2 o [f1,f2] = f2

P3<sub>23</sub>: DEF o [f1,f2,f3] :: ?2 o [f1,f2,f3] = f2

P3<sub>24</sub>: DEF o [f1,f2,f3,f4] :: ?2 o [f1,f2,f3,f4] = f2

(pour chaque arité de [...]) et pour tous les sélecteurs).

REMARQUE 1

Nous ne considérons pas d'axiomes sur "!" comme par exemple

f o !<sup>-</sup> = !<sup>-</sup> o f = !<sup>-</sup>

[f1,...,fi-1,!<sup>-</sup>,...,fn] = !<sup>-</sup>

?i o [f1,...,fn] = !<sup>-</sup> pour i>n.

Nous considérons toujours que les expressions qui codent des programmes itératifs sont correctes c'est-à-dire n'auront pas systématiquement une erreur durant leur exécution. Ces axiomes ne nous semblent pas utiles pour prouver l'équivalence d'expressions qui ne seront pas dans la classe des expressions équivalentes à "!".

REMARQUE 2

J. BACKUS propose en outre la famille d'équations

[f1 o ?1, f2 o ?2, ..., fn o ?n] o [g1, g2, ..., gn]  
 = [f1 o g1, f2 o g2, ..., fn o gn]

Cette famille est inutile dans notre ensemble d'axiomes car elle est conséquence équationnelle de l'ensemble d'axiomes {E1,E2,P3}. En effet,

[f1 o ?1, f2 o ?2, ..., fn o ?n] o [g1, g2, ..., gn]

en appliquant l'axiome E2

première partie

$=_A [(f_1 \circ ?1) \circ [g_1, g_2, \dots, g_n],$   
 $(f_2 \circ ?2) \circ [g_1, g_2, \dots, g_n], \dots, (f_n \circ ?n) \circ [g_1, g_2, \dots, g_n]]$

en appliquant l'axiome E1 "n fois"

$=_A [f_1 \circ (?1 \circ [g_1, g_2, \dots, g_n]),$   
 $f_2 \circ (?2 \circ [g_1, g_2, \dots, g_n]), \dots, f_n \circ (?n \circ [g_1, g_2, \dots, g_n])]$

et enfin, en appliquant l'axiome P3 "n fois"

$=_A [f_1 \circ g_1, f_2 \circ g_2, \dots, f_n \circ g_n]$

### 3.2. Des axiomes sur "ALL"

**Axiome de composition de deux fonctions "ALL f" et "ALL g"**

ALLALL:  $ALL f \circ ALL g = ALL (f \circ g)$   
 ~~~~~

Cet axiome est cité par J. BACKUS. Il correspond à la possibilité de composer entre elles deux itérations codées par les fonctions "ALL f" et "ALL g"

**Axiomes de "fusion" de n fonctions "ALL fi"**

FALL:  $trans \circ [ALL f_1, ALL f_2, \dots, ALL f_n] = ALL [f_1, f_2, \dots, f_n]$   
 ~~~~~

FALL correspond à une infinité d'axiomes:

$FALL_2: trans \circ [ALL f_1, ALL f_2] = ALL [f_1, f_2]$

$FALL_3: trans \circ [ALL f_1, ALL f_2, ALL f_3] = ALL [f_1, f_2, f_3] \dots$

FALL correspond à la possibilité de "fusionner" des itérations séparées codées par des fonctions "ALL fi" en une seule.

EXEMPLE

première partie

Nous utilisons les axiomes ALLALL et FALL pour montrer que les expressions (1) et (2) suivantes sont A-égales.

(1)  $ALL mult \circ (trans \circ [ALL a \circ (iota \circ n), ALL b \circ (iota \circ n)])$

(2)  $ALL (mult \circ [a, b]) \circ (iota \circ n)$

(1) et (2) ont toutes deux pour résultat la séquence des produits

$\langle a_1 b_1, a_2 b_2, \dots, a_n b_n \rangle$ .

On applique E2 à (1)

(1)  $=_A ALL mult \circ (trans \circ ([ALL a, ALL b] \circ (iota \circ n)))$

puis E1

(1)  $=_A ALL mult \circ ((trans \circ [ALL a, ALL b]) \circ (iota \circ n))$

ce qui permet d'appliquer FALL

(1)  $=_A ALL mult \circ (ALL [a, b] \circ (iota \circ n))$

on applique E2

(1)  $=_A (ALL mult \circ ALL [a, b]) \circ (iota \circ n)$

ce qui permet d'appliquer ALLALL et

(1)  $=_A (2)$

**Axiomes avec les primitives**

LALL:  $DEF \circ (ALL f) :: last \circ ALL f = f \circ last$   
 ~~~~~

HALL:  $DEF \circ (ALL f) :: hd \circ ALL f = f \circ hd$   
 ~~~~~

Il permettent de supprimer une séquence résultat de "ALL f".

EXEMPLE

On montre que les termes (1) et (2) de l'exemple 3 du chapitre I sont

première partie

A- égaux.

(1) last o (ALL / o (trans o [u,v]))

(2) / o (last o (trans o [u,v]))

En effet, en appliquant E1

(1) =<sub>A</sub> (last o ALL / ) o (trans o [u,v])

en appliquant LALL

(1) =<sub>A</sub> ( / o last) o (trans o [u,v])

et en appliquant E1

(1) =<sub>A</sub> (2)

TLALL: DEF o ALL f :: tl o ALL f = ALL f o tl  
 ~~~~~

L'axiome TLALL indique que tl est une fonction permutable avec "ALL f".

On a "tl o ALL f ⊂ ALL f o tl" et non l'inverse.

ALLID: ALL id = id  
 ~~~~~

ALLID place "ALL id" dans la classe de l'élément neutre id.

J. BACKUS donne encore

ALLVID: ALL f o <> = <>

ALLCONS: cons o [f o x, ALL f o a] = ALL f o (cons o [x,a]).

REMARQUE

D'autres axiomes mettant en jeu une fonction "ALL f" et des primitives sont cités dans l'annexe I.

### 3.3. Axiomes sur "\*"

Axiome de composition entre "\*" et "ALL g"

première partie

\*ALL: \*f o [x, ALL g o h] = \*(f o [?1, g o ?2]) o [x,h]  
 ~~~~~

\*ALL correspond à la possibilité de composer une fonction "ALL g" avec une fonction "\*f".

EXEMPLES

On montre que les termes (1) et (2) de l'exemple 1 du chapitre I sont A-égaux:

(1) last o ( \*mult o [1<sup>-</sup>, ALL 2<sup>-</sup> o (iota o n)])

(2) last o ( \*(mult o [?1,2<sup>-</sup>]) o [1<sup>-</sup>, iota o n] )

En effet, en appliquant \*ALL

(1) =<sub>A</sub> last o (\*(mult o [?1,2<sup>-</sup> o ?2]) o [1<sup>-</sup>,iota o n])

et en appliquant E3

(1) =<sub>A</sub> (2)

On montre exactement de la même manière que les termes (1) et (3) de l'exemple 2 du chapitre I sont A-égaux.

(1) last o ( \*mult o [1<sup>-</sup>, ALL ?1 o (dist1 o [x,iota o n]) ] )

(3) last o ( \*(mult o [?1,?1o?2]) o [1<sup>-</sup>,dist1 o [x,iota o n]) ] )

Axiomes de "fusion" entre n fonctions "\*fi"

F\*: trans o [ \*f1 o [x<sub>1</sub>,a] , \*f2 o [x<sub>2</sub>,a] , ..., \*fn o [x<sub>n</sub>,a] ]

~~~~~

= \*{ f1 o [?1o?1,?2]

~~~~~

, f2 o [?2o?1,?2],...

~~~~~

, fn o [?no?1,?2] ] o [ [x<sub>1</sub>,x<sub>2</sub>,...,x<sub>n</sub>] , a ]

~~~~~

Première partie

F\* correspond à une infinité d'axiomes:

F\*<sub>2</sub>: trans o [\*f1 o [x<sub>1</sub>,a],\*f2 o [x<sub>2</sub>,a]]  
 = [\*f1 o [?1o?1,?2],f2 o [?2o?1,?2]]  
     o [[x<sub>1</sub>,x<sub>2</sub>],a],  
 F\*<sub>3</sub>: trans o [\*f1 o [x<sub>1</sub>,a],f2 o [x<sub>2</sub>,a],f3 o [x<sub>3</sub>,a]]  
 = [\*f1 o [?1o?1,?2],f2 o [?2o?1,?2],f3 o [?3o?1,?2]]  
     o [[x<sub>1</sub>,x<sub>2</sub>,x<sub>3</sub>],a], ...

F\* permet de "fusionner" des itérations séparées codées par des fonctions "\*fi" en une seule itération. La séquence argument des "\*fi" est le résultat d'une même fonction a.

#### EXEMPLE

On montre que les expressions (2) et (5) de l'exemple 3 du chapitre I sont A-égales

(2) / o (last o (trans o [u,v]))  
     where def u = \*(mult o [?1,2<sup>-</sup>]) o [1<sup>-</sup>,iota o n]  
     and def v = \*mult o [1<sup>-</sup>,iota o n]  
 (5) / o (last o ( \*g o [ [1<sup>-</sup>,1<sup>-</sup>],iota o n ] ) )  
     where def g = [ mult o [2<sup>-</sup>,?1o?1], mult o [?2o?1,?2] ]

En appliquant F\*, on trouve

(2) =<sub>A</sub> / o (last o ( [\*f1,f2] o [[1<sup>-</sup>,1<sup>-</sup>],iota o n]))  
 pour f1 = (mult o [?1o?1,2<sup>-</sup>]) o [?1o?1,?2]  
 et f2 = mult o [?2o?1,?2]

en appliquant E1 à f1, on a

première partie

f1 =<sub>A</sub> mult o ([?1,2<sup>-</sup>] o [?1o?1,?2])

avec E2, on obtient

f1 =<sub>A</sub> mult o [?1 o [?1o?1,?2],2<sup>-</sup> o [?1o?1,?2]]

puis avec P3 et E3

f1 =<sub>A</sub> mult o [?1o?1,2<sup>-</sup>]

Si on a l'axiome MULTCOM de commutativité de la multiplication

(2) =<sub>A</sub> (5)

**Axiome de "fusion" pour composer "ALL g" et "\*f"**

FALL\*: trans o [ ALL g o (\*f o [x,h]) , \*f o [x,h] ]  
 =====  
 = \* ([g o f,f] o [?2o?1,?2]) o [[g o x,x],h]  
 =====

La fonction composée de "ALL g" avec "\*f" se fusionne avec "\*f". Nous pouvons utiliser \*ALL pour composer "\*f" avec "ALL g" et FALL\* pour composer "ALL g" avec "\*f".

**Axiome pour composer "\*g" avec "\*f"**

F\*: trans o [ \*g o [y, t1 o (\*f o [x,h]) ] , \*f o [x,h] ]  
 =====  
 = \* ([g,?2] o [?1o?1, f o [?2o?1,?2] ) o [[y,x],h]  
 =====

Pour composer "\*g" avec "\*f", on élimine l'initialisation x par application de la primitive "t1". Le résultat de cette composition est fusionné avec "\*f".

**Des axiomes de "fusion" et "composition" entre n fonctions "\*fi" ou "ALL fi"**

première partie

On peut combiner les axiomes de fusion. On obtient alors une famille infinie d'axiomes  $F$  qui contient la famille  $F^*$  et les axiomes  $FALL^*$  et  $F^{**}$ . La famille des axiomes  $F$  figure dans l'annexe I.

Avec  $ALLALL$ ,  $*ALL$ ,  $FALL$  et les axiomes  $F$ , nous avons mis dans  $A$  les axiomes permettant de montrer que deux expressions qui ne diffèrent que par des fusions et compositions d'itérations codées par les fonctions " $*f$ " et " $ALL\ g$ " sont  $A$ -égales.

#### Axiomes avec les primitives

$H^*$ :  $DEF\ o\ (*f\ o\ [x,a]) :: hd\ o\ (*f\ o\ [x,a]) = x$   
 ~~~~~

$TL^*$ :  $DEF\ o\ (tl\ o\ a) :: tl\ o\ (*f\ o\ [x,a]) = *f\ o\ [f\ o\ [x,a], tl\ o\ a]$   
 ~~~~~

$TL^*?l$ :  $tl\ o\ *?l = ALL\ ?l\ o\ distl$   
 ~~~~~

Si la longueur de la séquence argument de  $*?l$  est  $n$ , " $*?l$ " donne une séquence de  $x$  de longueur  $n$ .

$*?2$ :  $*?2 = cons$   
 ~~~~~

$*VID$ :  $*f\ o\ [z, <>] = cons\ o\ [z, <>]$   
 ~~~~~

$*CONS$ :  $cons\ o\ [z, *f\ o\ [f\ o\ [z,x],a]] = *f\ o\ [z, cons\ o\ [x,a]]$   
 ~~~~~

#### REMARQUE

D'autres axiomes mettant en jeu une fonction  $*f$  et des primitives sont citées dans l'annexe I.

première partie

#### 3.4. Axiomes sur " $WH$ "

Nous avons des axiomes de composition analogues à ceux de " $*$ ".

#### Axiome pour composer " $WH(p,f)$ " avec " $ALL\ g$ "

$WHALL$ :  $WH(p,f)\ o\ [x, ALL\ g\ o\ h]$   
 ~~~~~

$= WH(p\ o\ [?l, g\ o\ ?2], f\ o\ [?1, g\ o\ ?2])\ o\ [x, h]$   
 ~~~~~

#### Les axiomes de "fusion et composition" particuliers

Dans le cas général, il n'est pas possible de fusionner des fonctions " $WH(pi, fi)$ ". Les séquences résultats sont de longueur différentes à cause des prédicats  $pi$  et on ne peut leur appliquer  $trans$ .

Cependant, on peut trouver des cas particuliers où la fusion est possible.

#### Quand le prédicat s'applique uniquement aux éléments de la séquence

C'est le cas pour  $n$  fonctions " $WH(p, fi)$ " à condition que le prédicat  $p$  ne s'applique qu'aux éléments de la séquence commune qui figure dans l'argument de ces  $n$  fonctions. Par exemple

$trans\ o\ [WH(p\ o\ ?2, f)\ o\ [x, h], WH(p\ o\ ?2, g)\ o\ [y, h]]$

$=_{ext}\ WH(p\ o\ ?2, [f\ o\ [?1o?1, ?2], g\ o\ [?2o?2]])\ o\ [[x, y], h]$

On montre que ces termes sont  $A$ -égaux en utilisant l'axiome  $F^*$  et l'axiome

Première partie

WH\*PART: DEF o (WH(p o ?2,?2) o [y,a]) ::  
 ~~~~~

\*f o [x,t1 o (WH(p o ?2,?2) o [y,a])] = WH(p o ?2,f) o [x,a]  
 ~~~~~

Axiomes pour composer "ALL g" avec "WH(p,f)"

C'est aussi possible de fusionner "WH(p,f)" avec "ALL g o WH(p,f)"

ALLWH: trans o [ ALL g o ( WH(p,f) o [x,h] ) , WH(p,f) o [x,h] ]  
 ~~~~~  
 = WH ( p o [?2o?1,?2] , [g o f,f] o [?2o?1,?2] ) o [[g o x,x],h]  
 ~~~~~

On a aussi l'axiome "symétrique" que l'on a répertorié ALLWHS dans l'annexe I.

Axiomes pour composer "\*f" avec "WH(p,g)"

On peut encore fusionner

"WH(p,g) o [y,h]" avec "\*f o [x,t1 o (WH(p,g) o [y,h])]"

\*WH: trans o [ \*f o [ x , t1 o ( WH(p,g) o [y,h] ) ] ,  
 ~~~~~  
 WH(p,g) o [y,h] ]  
 ~~~~~  
 = WH ( p o [?2o?1,?2] , [f,?2] o [?1o?1,g o [?2o?1,?2]] )  
 ~~~~~  
 o [[x,y],h]  
 ~~~~~

L'axiome "symétrique" \*WHS figure dans l'annexe I.

Autres axiomes de composition

première partie

Axiomes pour composer "WH(p,g)" avec "\*f"

On ne peut pas écrire ces axiomes en utilisant "trans" comme nous l'avons fait avec "\*" puisque les séquences résultats de

WH(p,g) o [x,t1 o (\*f o [y,h])]

et de \*f o [y,h]

ne sont pas forcément de même longueur. Cependant, on peut utiliser une composition par "ALL ?1" ou "ALL ?2". On comprendra cela plus facilement si l'on sait que, dans le cas où "a:x" et "b:x" ont pour résultat deux séquences non vides de même longueur, l'on a

(ALL?1 o (trans o [a,b]):x == a:x

et (ALL?2 o (trans o [a,b]):x == b:x.

On peut voir un axiome de composition WH\* avec "ALL?1" et un axiome WH\*S avec "ALL?2" dans l'annexe I.

Axiomes pour composer "WH(p,g)" avec "WH(q,f)"

On doit introduire une conditionnelle et non un "and" pour composer les prédicats d'arrêt. C'est un schéma bien connu des programmeurs pour les boucles. On a ainsi deux axiomes de composition WHWH avec "ALL?1" et WHWHS avec "ALL?2" présentés dans l'annexe I.

Remarquons que maintenant, nous avons dans A le catalogue complet des axiomes de fusion et de composition des fonctions "ALL f", "\*f" et "WH(p,f)" qui codent les itérations.

Axiomes avec les primitives sur les séquences

Il reste à ajouter des axiomes entre "WH" et les primitives sur les

première partie

séquences qui sont présentés dans l'annexe I. Considérons par exemple l'axiome TLWH

TLWH:  $p \circ [x, hd \circ a] ::$

$tl \circ (WH(p, f) \circ [x, a]) = WH(p, f) \circ [f \circ [x, hd \circ a], tl \circ a]$

On peut constater que la précondition n'est pas de la forme "DEF o t". Pour appliquer cet axiome, il faut prouver la précondition (voir le paragraphe 2.4) soit

$p \circ [x, hd \circ a] =_{ext(ouA)} I^-$

### 3.5. Axiomes pour les primitives sur les séquences

#### Axiomes sur "trans"

Au paragraphe 2.3.b), nous avons prouvé la validité de:

LTRANS: DEF o (trans o [a, b])

$:: last \circ (trans \circ [a, b]) = [last \circ a, last \circ b]$

HTRANS: DEF o (trans o [a, b])

$:: hd \circ (trans \circ [a, b]) = [hd \circ a, hd \circ b]$

La primitive "tl" permute avec "trans"

TLTRANS:  $tl \circ (trans \circ [a, b]) = trans \circ [tl \circ a, tl \circ b]$ .

Nous avons des familles infinies d'axiomes correspondant aux diverses

arités de "trans". Les axiomes LTRANS, HTRANS et TLTRANS sont les éléments correspondant à l'arité 2. Ces familles sont répertoriées dans l'annexe I.

Et enfin une famille d'axiomes permet de supprimer "trans":

ALL?1TRANS<sub>12</sub>: DEF o (trans o [a, b]) ::

$ALL?1 \circ (trans \circ [a, b]) = a$

ALL?2TRANS<sub>22</sub>: DEF o (trans o [a, b]) ::

$ALL?2 \circ (trans \circ [a, b]) = b$

La famille infinie de ces axiomes est présentée dans l'annexe I dans le paragraphe des axiomes sur "ALL". Ces axiomes permettent d'appliquer les compositions de "ALL" avec "\*", "\*" avec "\*", ... à partir des axiomes de fusion et composition FALL\*, F\*\*,...

EXEMPLE UTILISANT FALL\*, LALL et ALL?1TRANS

On peut montrer que les expressions (1) et (3) de l'exemple 7 du chapitre I sont A-égales.

(1)  $?1 \circ (last \circ (*f \circ [[0^-, 0^-], x]))$

where def f =  $[ / \circ [g, 10^-], g ]$

where def g =  $+ \circ [?2 \circ ?1, ?2]$

(3)  $/ \circ [last \circ (*+ \circ [0^-, x]), 10^-]$

En appliquant E3 et P1, on a

$$f =_A [ / o [id o g, 10^- o g] , g ]$$

on peut appliquer E2 à f:

$$f =_A [ / o ((id, 10^-) o g , g ]$$

Après avoir substitué l'expression de g, on applique E1 et E2

$$f =_A [ ( / o [id, 10^-] ) o + , + ] o [?2o?1, ?2]$$

Si l'on sait que

$$+ o [0^-, 0^-] =_A 0^-,$$

on trouve

$$[0^-, 0^-] =_A [+ o [0^-, 0^-], 0^-].$$

Après avoir substitué l'expression de f, on peut alors appliquer FALL\*

dans l'expression (1) à

$$t = *f o [+ o [0^-, 0^-], 0^-]$$

et on trouve

$$(1) =_A ?1 o (last o t),$$

$$\text{pour } t = \text{trans } o [ALL / o (*+ o [0^-, x]), *+ o [0^-, x]]).$$

Si on applique E1 puis LALL, on a

$$(2) =_A \text{last } o (ALL?1 o t)$$

en appliquant alors ALL?1TRANS et on trouve

$$(2) =_A (3).$$

**Axiomes sur la primitive "distl"**

L'axiome suivant a été prouvé au paragraphe 2.2.b).

première partie

$$\text{LDISTL: last } o (\text{distl } o [x, a]) = [x, \text{last } o a]$$

On a aussi un axiome avec hd

$$\text{HDISTL: hd } o (\text{distl } o [x, a]) = [x, \text{hd } o a]$$

et un axiome avec tl

$$\text{TLDISTL: tl } o (\text{distl } o [x, a]) = \text{distl } o [x, \text{tl } o a]$$

On présente à la fin du paragraphe des axiomes sur "ALL" dans l'annexe I un axiome particulier avec "ALL"

$$\begin{aligned} \text{ALLDISTL: ALL } [f o ?1, g o ?2] o (\text{distl } o [x, a]) \\ = \text{distl } o [f o x, \text{ALL } g o a] \end{aligned}$$

On a aussi un axiome entre "trans" et "distl"

$$\text{TDISTL: trans } o [\text{ALL?1 } o (\text{distl } o [x, a]), a] = \text{distl } o [x, a]$$

EXEMPLES POUR ILLUSTRER LE ROLE DES AXIOMES ALLODISTL ET TDISTL

Ces deux axiomes sont utilisés pour prouver l'équivalence entre deux expressions  $e_1$  et  $e_2$  qui codent des programmes itératifs qui ne diffèrent que par la place d'une expression z. z est sous-terme de f dans l'itération " $*f o [x, a]$ " pour  $e_1$  et z est située dans l'argument " $[x, \text{distl } o [z, a]]$ " de l'itération correspondante pour  $e_2$ .

Premier exemple avec ALLODISTL

Cet axiome est utilisé pour montrer que les expression (2) et (3) de l'exemple 2 du chapitre I sont A-égales.

Première partie

$$(2) = \text{last } o (*g \circ [1^-, \text{distl } o [\text{id}, \text{iota } o n]])$$

$$\text{where def } g = \text{mult } o [?1, x \circ (?1o?2)]$$

$$(3) = \text{last } o (*g' \circ [1^-, \text{distl } o [x, \text{iota } o n]])$$

$$\text{where def } g' = \text{mult } o [?1, ?1o?2]$$

Nous pouvons remarquer les places respectives de  $x$  dans (2) et (3)

Soit  $a = \text{distl } o [x, \text{iota } o n]$ ,

sachant qu'en appliquant P1:  $x =_A x \circ \text{id}$

et qu'en appliquant ALLID:

$\text{iota } o n =_A \text{ALL id } o (\text{iota } o n)$ , on a

$a =_A \text{distl } o [x \circ \text{id}, \text{ALL id } o (\text{iota } o n)]$

et que, si on applique ALLDISTL

$a =_A \text{ALL}[x \circ ?1, \text{id } o ?2] \circ (\text{distl } o [\text{id}, \text{iota } o n])$

en appliquant P1

$a =_A \text{ALL}[x \circ ?1, ?2] \circ (\text{distl } o [\text{id}, \text{iota } o n])$ .

Si l'on substitue l'expression de  $g'$  dans l'expression (3), on applique

\*ALL et on trouve

$(3) =_A \text{last } o (*\text{mult } o [1^-, \text{ALL} ?1 \circ a])$ .

Maintenant en considérant l'expression de  $a$ , on peut appliquer ALLALL

$(3) =_A \text{last } o$

$(*\text{mult } o [1^-, \text{ALL}(?1 \circ [x \circ ?1, ?2]) \circ (\text{distl } o [\text{id}, \text{iota } o n])])$

puis avec P3

$(3) =_A \text{last } o$

$(*\text{mult } o [1^-, \text{ALL}(x \circ ?1) \circ (\text{distl } o [\text{id}, \text{iota } o n])])$

et enfin en utilisant \*ALL et E1, on trouve

première partie

$(3) =_A (2)$

Le deuxième exemple utilise aussi ALLDISTL.

ALLDISTL est utilisé pour montrer que les expressions (2) et (3) de l'exemple 6 du chapitre I sont A-égales.

$(2) = \text{ALL}(/ \circ [?2, ?1]) \circ (\text{distl } o [s_2, \text{ALL } s_1 \circ (\text{iota } o n)])$

$(3) = \text{ALL}(/ \circ [s_1 \circ ?2, ?1]) \circ (\text{distl } o [s_2, \text{iota } o n])$

On peut noter la place différente de  $s_1$  dans (2) et (3).

Soit  $a = \text{distl } o [s_2, \text{ALL } s_1 \circ (\text{iota } o n)]$

en appliquant P1:  $s_2 =_A \text{id } o s_2$ ,

on peut alors appliquer ALLDISTL puis P1

$a =_A \text{ALL}[?1, s_1 \circ ?2] \circ (\text{distl } o [s_2, \text{iota } o n])$ .

Dans l'expression (2) on peut alors appliquer E1 puis ALLALL

$(2) =_A \text{ALL}(/ \circ [?2, ?1]) \circ [?1, s_1 \circ ?2] \circ (\text{distl } o [s_2, \text{iota } o n])$

en appliquant alors E2 et P3

$(2) =_A (3)$ .

Le troisième exemple nécessite l'application de IDISTL

Maintenant, on veut montrer que les expressions (1) et (3) de l'exemple 6 du chapitre I sont A-égales.

$(1) \text{ALL}(/ \circ [s_1, s_2]) \circ (\text{iota } o n)$

$(3) \text{ALL}(/ \circ [s_1 \circ ?2, ?1]) \circ (\text{distl } o [s_2, \text{iota } o n])$ .

Nous pouvons remarquer la place de  $s_2$  dans chaque expression.

première partie

En appliquant ALLALL dans l'expression (3)

$$(3) =_A (ALL / o \text{ ALL}[s_1 o ?2, ?1]) o (\text{distl } o [s_2, \text{iota } o n])$$

puis avec FALL

$$(3) =_A (ALL / o (\text{trans } o [ALL(s_1 o ?2), ALL?1])) o (\text{distl } o [s_2, \text{iota } o n])$$

avec E1 puis E2:

$$(3) =_A ALL / o (\text{trans } o [u, v])$$

pour  $u = ALL(s_1 o ?2) o (\text{distl } o [s_2, \text{iota } o n])$   
et  $v = ALL?1 o (\text{distl } o [s_2, \text{iota } o n])$ .

Considérons déjà le terme u

On applique ALLALL:

$$u =_A (ALL s_1 o ALL?2) o (\text{distl } o [s_2, \text{iota } o n])$$

Si  $k = \text{distl } o [s_2, \text{iota } o n]$  en appliquant TDISTL on a

$$k =_A \text{trans } o [ALL?1 o (\text{distl } o [s_2, \text{iota } o n]), \text{iota } o n]$$

donc avec E1 puis ALL?2TRANS

$$u =_A ALL s_1 o (\text{iota } o n).$$

Considérons maintenant v

$$\text{Soit } h = \text{distl } o [s_2, \text{iota } o n]$$

On applique P1  $s_2 =_A s_2 o \text{id}$ ,

$$\text{et } ALLID \text{ iota } o n =_A ALL \text{id } o (\text{iota } o n)$$

première partie

on peut alors appliquer ALLDISTL à h

$$h =_A ALL[s_2 o ?1, \text{id } o ?2] o (\text{distl } o [\text{id}, \text{iota } o n]).$$

On applique maintenant à  $v = ALL?1 o h$  les axiomes E1, ALLALL et P3

$$v =_A ALL(s_2 o ?1) o (\text{distl } o [\text{id}, \text{iota } o n]).$$

Maintenant il nous faut considérer l'expression de  $s_2$

$$s_2 = \text{last } o (*+ o [0^-, ALL p o (\text{iota } o 10^-)]).$$

Qu'on peut aussi écrire avec application de P1, E3, E1 et E2

$$s_2 =_A s o 0^-$$

$$\text{pour } s = \text{last } o (*+ o [\text{id}, ALL p o (\text{iota } o 10^-)])$$

on a donc

$$v =_A ALL ((s o 0^-) o ?1) o (\text{distl } o [\text{id}, \text{iota } o n])$$

avec E1 et E3

$$v =_A ALL (s o 0^-) o (\text{distl } o [\text{id}, \text{iota } o n]).$$

Il est alors possible en appliquant E3 d'obtenir

$$v =_A ALL (s o (0^- o ?2)) o (\text{distl } o [\text{id}, \text{iota } o n]).$$

Comme on peut toujours appliquer TDISTL à  $h = \text{distl } o [\text{id}, \text{iota } o n]$

$$b =_A \text{trans } o [ALL?1 o (\text{distl } o [\text{id}, \text{iota } o n]), \text{iota } o n],$$

il suffit d'appliquer E1 et ALLALL pour avoir

première partie

$$v =_A \text{ALL} (s \circ 0^-) \circ (\text{ALL} ?2 \circ b)$$

et d'appliquer ALL?2TRANS et " $s \circ 0^- = s_2$ " pour obtenir

$$v =_A \text{ALL} s_2 \circ (\text{iota} \circ n).$$

Revenons à l'expression (3)

$$(3) =_A \text{ALL} / \circ (\text{trans} \circ [\text{ALL} s_1 \circ (\text{iota} \circ n), \text{ALL} s_2 \circ (\text{iota} \circ n)])$$

il reste à appliquer FALL pour obtenir

$$(3) =_A (1).$$

Axiome sur "last" et "tl"

LTL: DEF  $\circ (\text{last} \circ \text{tl}) :: \text{last} \circ \text{tl} = \text{last}$   
 ~~~~~

LTL permet de supprimer "tl".

Axiomes sur "iota"

Citons

LIOTA: DEF  $\circ \text{iota} :: \text{last} \circ \text{iota} = \text{id}$   
 ~~~~~

ALL\*IOTA:  $\text{ALL}(\text{last} \circ (*f \circ [g \circ x^-, \text{iota}])) \circ \text{iota}$   
 ~~~~~  
 $= \text{tl} \circ (*f \circ [g \circ x^-, \text{iota}]).$   
 ~~~~~

Ce dernier axiome montre le cas où l'on peut supprimer une "imbrication d'itération".

EXEMPLE DE SUPPRESSION D'IMBRICATION AVEC ALL\*IOTA

première partie

Nous allons pouvoir montrer qu'une expression de calcul de

$$1+2+4+\dots+2^n$$

(qui ne contient pas d'itération imbriquée) est A-égale à l'expression (i) de l'exemple 5 du chapitre I.

$$(i) \text{ last} \circ (*g \circ [0^-, \text{iota} \circ n])$$

where def  $g = + \circ [?1, f \circ ?2]$

where def  $f = \text{last} \circ (*\text{mult} \circ [1^-, \text{ALL} 2^- \circ \text{iota}])$

f code une itération "imbriquée" dans l'itération "g".

Après avoir substitué l'expression de g, on applique \*ALL et

$$(i) =_A \text{last} \circ (*+ \circ [0^-, \text{ALL} f \circ (\text{iota} \circ n)]).$$

Après avoir substitué l'expression de f, on peut appliquer ALL\*IOTA

$$(i) =_A \text{last} \circ (*+ \circ [0^-, \text{tl} \circ (*\text{mult} \circ [1^-, \text{ALL} 2^- \circ (\text{iota} \circ n)])]).$$

Cette expression ne contient plus d'itération "imbriquée"

EXEMPLE UTILISANT LIOTA ET LTL

Nous prouvons que les expressions (2) et (3) de l'exemple 7 chapitre I sont A-égales.

$$(2) \text{ last} \circ (\text{ALL} / \circ u)$$

where def  $u = \text{trans} \circ [\text{tl} \circ (*+ \circ [0^-, x]), \text{iota} \circ 10^-]$

$$(3) / \circ [\text{last} \circ (*+ \circ [0^-, x]), 10^-]$$

Première partie

On applique E1 puis LALL puis E1 à (2)

$$(2) =_A / o \text{ (last } o \text{ u)}.$$

On applique LTRANS

$$(2) =_A / o \text{ [last } o \text{ (tl } o \text{ (* } o \text{ [0}^-, x]), last } o \text{ (iota } o \text{ 10}^-)].$$

On applique alors LTL et LIOTA

$$(2) =_A (3)$$

REMARQUE

L'annexe I propose d'autres axiomes sur ces primitives. En particulier, on y trouve les axiomes concernant les primitives "cons" et "append".

### 3.6. Axiomes sur " $\rightarrow$ ;"

Axiomes avec " $\circ$ "

$$\text{COND1: } (p \rightarrow f; g) \circ h = (p \circ h) \rightarrow (f \circ h); (g \circ h)$$

$$\text{COND2: } h \circ (p \rightarrow g; h) = p \rightarrow (h \circ f); (h \circ g)$$

COND2 est cité par Z. MANNA [Man73]. J. BACKUS les donne tous les deux.

Axiomes qui suppriment un test

$$\text{COND3: } p \rightarrow (p \rightarrow f; g); h = p \rightarrow f; h$$

$$\text{COND4: } p \rightarrow h; (p \rightarrow f; g) = p \rightarrow h; g.$$

Axiomes avec " $[_\dots]$ "

première partie

$$\text{COND5: } [h, p \rightarrow f; g] = p \rightarrow [h, f]; [h, g]$$

$$\text{COND6: } [p \rightarrow f; g, h] = p \rightarrow [f, h]; [g, h].$$

Ces axiomes font partie d'une famille infinie citée par J. BACKUS

$$\begin{aligned} \text{COND5}_i: & [h_1, \dots, h_{i-1}, p \rightarrow f; g, h_{i+1}, \dots, h_n] \\ & = p \rightarrow [h_1, \dots, h_{i-1}, f, h_{i+1}, \dots, h_n]; [h_1, \dots, h_{i-1}, g, h_{i+1}, \dots, h_n]. \end{aligned}$$

Axiomes pour " $\rightarrow$ ;"

$$T: (p \rightarrow x; y) \rightarrow f; g = p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)$$

$$C: p \rightarrow (q \rightarrow f; g); (q \rightarrow x; y) = q \rightarrow (p \rightarrow f; x); (p \rightarrow g; x)$$

Les axiomes C, T, COND3, COND4 sont donnés par S. BLOOM et R. TINDELL [Blo83].

Cet ensemble d'axiomes permet de supprimer les redondances de test illustrées par les exemples 8 et 9 du chapitre I.

EXEMPLE 8 DU CHAPITRE I

$$(e) \quad p_1 \rightarrow r; (p_2 \rightarrow s; (p_3 \rightarrow t; u))$$

$$\text{avec } t = q_2 \rightarrow u; (p_3 \rightarrow k; z).$$

On montre déjà que

$$u =_A p_3 \rightarrow u; u$$

pour appliquer ensuite C à t ce qui permettra d'appliquer COND3 pour faire disparaître la redondance de p3.

première partie

On applique l'axiome P3

$$u =_A ?1 \text{ o } [u, p3 \rightarrow k; z].$$

On applique COND5:

$$u =_A ?1 \text{ o } (p3 \rightarrow [u, k]; [u, z])$$

et avec COND2

$$u =_A p3 \rightarrow (?1 \text{ o } [u, w]); (?1 \text{ o } [u, z])$$

alors P3 donne

$$u =_A p3 \rightarrow u ; u.$$

On applique C à t

$$t =_A p3 \rightarrow (q2 \rightarrow u; k) ; (q1 \rightarrow u; z)$$

alors COND3 s'applique à (e) et

$$(e) =_A p1 \rightarrow r ; (p2 \rightarrow s ; (p3 \rightarrow (q1 \rightarrow v; w) ; z)) \text{ (e}_1\text{)}.$$

Dans  $e_1$ , on ne trouve plus de redondance du test p3

EXEMPLE 9 DU CHAPITRE 1

$$(f) \text{ q1} \rightarrow ( \text{ p1} \rightarrow \text{ h} ; ( \text{ p2} \rightarrow \text{ m} ; \text{ u} ) ) \\ ; ( \text{ p1} \rightarrow \text{ n} ; ( \text{ p2} \rightarrow \text{ m} ; ( \text{ p3} \rightarrow \text{ k} ; \text{ u} ) ) ).$$

Dans (f) le test q1 est fait inutilement dans le cas du résultat m.

On peut appliquer C

première partie

$$(f) =_A \text{ p1} \rightarrow ( \text{ q1} \rightarrow \text{ h} ; \text{ n} ) ; ( \text{ q1} \rightarrow ( \text{ p2} \rightarrow \text{ m} ; \text{ u} ) \\ ; ( \text{ p2} \rightarrow \text{ m} ; ( \text{ p3} \rightarrow \text{ k} ; \text{ u} ) ) ).$$

On peut encore appliquer C

$$(f) =_A \text{ p1} \rightarrow ( \text{ q1} \rightarrow \text{ h} ; \text{ n} ) ; ( \text{ p2} \rightarrow ( \text{ q1} \rightarrow \text{ m} ; \text{ m} ) \\ ; ( \text{ q1} \rightarrow \text{ u} ; ( \text{ p3} \rightarrow \text{ k} ; \text{ u} ) ) ).$$

Maintenant il est simple de montrer comme pour l'exemple précédent que

$$\text{q1} \rightarrow \text{m} ; \text{m} =_A \text{m}$$

et l'on trouve

$$(f) =_A \text{ p1} \rightarrow ( \text{ q1} \rightarrow \text{ h} ; \text{ n} ) ; ( \text{ p2} \rightarrow \text{ m} \\ ; ( \text{ q1} \rightarrow \text{ u} ; ( \text{ p3} \rightarrow \text{ k} ; \text{ u} ) ) ).$$

Dans cette expression, q1 n'est pas testé dans le cas d'un résultat m.

REMARQUE 1

On peut aussi ajouter d'autres axiomes donnés par S.BLOOM et R.TINDELL [Blo83]

$$\text{TRUE: } T^- \rightarrow f; g = f$$

$$\text{FALSE: } F^- \rightarrow f; g = g$$

$$\text{PPX: } p \rightarrow p; x = p \rightarrow T^-; x$$

$$\text{PXP: } p \rightarrow x; p = p \rightarrow x; F^-.$$

Ces axiomes n'ont pas un grand intérêt si l'on n'inclut pas les axiomes sur les booléens.

REMARQUE 2

Pour les raisons que nous avons invoquées dans la remarque du paragraphe 3.1, on ne considère pas d'axiomes sur "!" comme par exemple

première partie

$p \rightarrow !\bar{;}!\bar{=} !\bar{}$   
 ou  $!\bar{=} \rightarrow f;g = !\bar{}$ .

#### 4. Théories particulières

Notre théorie A est incomplète. Nous avons été parfois tenus (comme dans les exemples des paragraphes 3.3 et 3.5) d'ajouter des axiomes concernant les primitives de base. Il faudrait donc inclure aussi des axiomes sur ces primitives de base. Nous n'estimons pas devoir en donner la liste complète car nous voulons rester dans le cadre "intrinsèque" de cette thèse. Nous signalons dans l'annexe I uniquement ceux que nous avons été amenés à utiliser pour traiter les exemples du chapitre I.

Cependant, des propriétés particulières à des primitives de base (comme par exemple l'associativité de "+" ou "mult") peuvent servir à prouver des propriétés particulières de fonctions "\*" ou "mult". Ces propriétés sont alors les mêmes pour toute fonction "\*f" qui concerne une fonction f associative. On a alors une sorte de règle d'inférence pour des preuves sur ces fonctions "\*f". Ce paragraphe est consacré à de telles règles d'inférence que nous pourrions appliquer pour prouver que des expressions sont A-égales. Nous pouvons alors montrer l'équivalence des expressions (1) et (4) de l'exemple 3 et des expressions (1) et (2) des l'exemple 4 et 5 du chapitre I. Nous savons que montrer une équivalence analogue dans le système de R. BURSTALL et J. DARLINGTON [Bur77] réclame la réponse à un eureka ou l'application d'heuristiques particulières.

#### 4.1. Première règle

##### PROPRIÉTÉ (PA)

Notons "." et ":" deux opérations binaires vérifiant la propriété suivante:

$$(PA(.,:)) \quad a.(b:c) = (a.b):c .$$

Les opérations associatives comme l'addition ou la multiplication dans les entiers ou les réels vérifient cette propriété avec elles mêmes.

Dans les entiers ou les réels, on a aussi  $PA(+,-)$

$$a + (b-c) = (a+b) - c .$$

Notons "\*" la multiplication et "/" la division dans les réels, on a encore  $PA(*, /)$

$$a*(b/c) = (a*b)/c .$$

Notons "\" l'opération telle que

$$a \backslash b = b/a,$$

on a  $PA(\backslash, *)$  car

$$a \backslash (b*c) = (b*c)/a = (b/a)*c = (a \backslash b)*c .$$

Considérons deux fonctions f et g qui correspondent à deux opérations ayant la propriété (PA), dans le formalisme FP cela s'écrit:

$$(PA(g,f)) \quad g \circ [a, f \circ [b, c]] = f \circ [g \circ [a, b], c].$$

~~~~~

Si nous avons dans A l'axiome d'associativité de "+"

Notons "\*" la multiplication et "/" la division dans les réels, on a

PLUSASS:  $+ \circ [a, + \circ [b, c]] = + \circ [+ \circ [a, b], c],$

on sait que  $PA(+, +)$ .

Si l'on a aussi l'axiome définissant "-"

MOINS:  $+ \circ [a, - \circ [0^-, b]] = - \circ [a, b],$

on peut montrer que  $PA(+, -)$ . En appliquant MOINS

$+ \circ [a, - \circ [b, c]] =_A + \circ [a, + \circ [b, - \circ [0^-, c]]]$

avec PLUSASS

$+ \circ [a, - \circ [b, c]] =_A + \circ [+ \circ [a, b], - \circ [0^-, c]]$

et avec MOINS

$+ \circ [a, - \circ [b, c]] =_A - \circ [+ \circ [a, b], c].$

On montre de même que  $PA(mult, mult)$  et  $PA(mult, /)$  en utilisant les axiomes MULTASS et DIV/.

MULTASS:  $mult \circ [a, mult \circ [b, c]] = mult \circ [mult \circ [a, b], c]$

DIV/:  $mult \circ [a, / \circ [1^-, b]] = / \circ [a, b].$

Mais on peut aussi appliquer d'autres axiomes de A pour montrer que deux fonctions ont la propriété (PA). Par exemple, avec la fonction

$e = / \circ [?2, ?1],$

on a  $PA(e, mult)$ . La fonction e correspond à l'opération notée "/" ci-dessus.

première partie

On applique E1 puis P3 et

$e \circ [a, mult \circ [b, c]] =_A / \circ [mult \circ [b, c], a]$

en appliquant alors MULTCOM axiome de commutativité de "mult",  $PA(mult, /)$  puis de nouveau MULTCOM

$=_A / \circ [mult \circ [c, b], a] =_A mult \circ [c, / \circ [b, a]] =_A mult \circ [/ \circ [b, a], c]$

on applique alors E1, E2 et P3

$=_A mult \circ [(/ \circ [?2, ?1]) \circ [a, b], c]$

et on a montré que  $PA(/ \circ [?1, ?2], mult)$ .

On voit donc que: en utilisant la A-égalité, on peut montrer que  $PA(g, f)$  pour deux fonctions f et g.

R1: REGLE UTILISANT LA PROPRIETE (PA)

$PA(g, f) \vdash \text{ALL } g \circ (dist1 \circ [x, *f \circ [z, a]]) = *f \circ [g \circ [x, z], a].$   
 ~~~~~

La justification se fait simplement par induction sur la structure de la séquence a.

Intuitivement si  $PA(., :)$ ,

$a.(b:c) = (a.b):c$  donc

$x.(((z:a1):a2):a3):a4 = (x.(((z:a1):a2):a3)):a4$

$= ((x.((z:a1):a2)):a3):a4 = (((x.z):a1):a2):a3):a4$

$= (((x.z):a1):a2):a3):a4$

et c'est la même chose pour un nombre i quelconque de ai.

première partie

EXEMPLE D'APPLIQUATION: SIMPLIFICATION DU CALCUL DE  $n!/p!$ .

Considérons l'expression

(e)  $/ o [last o u, v]$   
 where def  $u = *mult o [v, s]$

and def  $v = last o (*mult o [1^-, iota o p])$ .

La fonction  $v$  calcule  $p!$ . Si la fonction  $s$  a pour résultat la séquence  $\langle p+1, p+2, \dots, n+p-1 \rangle$ , la fonction  $u$  calcule  $n!$  en utilisant  $v$ . (e) calcule  $n!/p!$ .

REMARQUE

On peut aussi partir de l'expression directe de  $n!$

$h = *mult o [1^-, iota o n]$

et montrer que  $h$  et  $u$  sont A-égales en utilisant les axiomes APPIOTA et \*APP présentés dans l'annexe I.

Nous allons simplifier l'expression  $e$

En utilisant P3 et E2, on montre que

$(e) =_A / o ([?2, ?1] o [v, last o u])$

on applique alors LDIS.L

$(e) =_A / o ([?2, ?1] o (last o (dist1 o [v, u])))$ .

Et en utilisant E1 et LALL

$(e) =_A last o (ALL (/ o [?2, ?1]) o (dist1 o [u, v]))$ .

première partie

On peut alors appliquer la règle R1 puisque  $PA(/ o [?2, ?1], mult)$ :

$(e) =_A last o (*mult o [ (/ o [?2, ?1]) o [v, v], s])$ .

On applique alors E1, E2 et P3

$(e) =_A last o (*mult o [ (/ o [v, v]), s])$ .

Et il suffit d'appliquer l'axiome IDIV/

$IDIV/:: DEF o (/ o [x, x]) :: / o [x, x] = 1^-$

pour avoir

$(e) =_A last o (*mult o [1^-, s])$

et on a montré que  $n!/p!$  peut se calculer par le produit

$(\dots((1*(p+1))*(p+2))*\dots)*(n+p-1)$

#### 4.2. Deuxième règle

PROPRIETE (PB)

Notons "." et ":" deux opérations binaires ayant la propriété suivante

$(PB(., :)) (x.y):(a.b) = (x:a).(y:b)$ .

Les opérations associatives et commutatives comme l'addition et la multiplication dans les entiers ou les réels vérifient la propriété (PB) avec elles-mêmes.

Dans les réels, on a aussi

première partie

$$PB(+,-) \quad (x + y) - (a + b) = (x - a) + (y - b)$$

$$PB(*,/) \quad (x * y) / (a * b) = (x / a) * (y / b).$$

Considérons deux fonctions  $f$  et  $g$  qui correspondent à deux opérations ayant la propriété (PB), elles vérifient l'équation

$$(PB(f,g)) \quad g \circ [f \circ \{x,y\}, f \circ \{a,b\}] = f \circ [g \circ \{x,a\}, g \circ \{y,b\}].$$

Avec les axiomes PLUSCOM et PLUSASS de commutativité et d'associativité de "+", on a

$$\begin{aligned} + \circ [+ \circ \{x,y\}, + \circ \{a,b\}] &=_A + \circ \{x, + \circ \{y, + \circ \{a,b\}\}\} \\ &=_A + \circ \{x, + \circ \{+ \circ \{a,b\}, y\}\} \\ &=_A + \circ \{x, + \circ \{a, + \circ \{b,y\}\}\} \\ &=_A + \circ \{+ \circ \{x,a\}, + \circ \{b,y\}\}. \end{aligned}$$

Donc  $PB(+,+)$ .

On vérifierait de même que  $PB(mult,mult)$  en utilisant les axiomes MULTCOM et MULTASS.

On montre que  $PB(-,+)$

$$+ \circ [- \circ \{x,y\}, - \circ \{a,b\}] =_A - \circ \{+ \circ \{x,a\}, + \circ \{y,b\}\},$$

en utilisant l'ensemble des axiomes de la structure de groupe pour l'opération "+" c'est-à-dire PLUSASS, PLUSCOM, PLUSO, MOINS et IMOINS.

$$PLUSO: + \circ [a, 0^-] = a$$

$$IMOINS: T^- \circ (- \circ [a,a]) :: - \circ [a,a] = 0^-.$$

En effet, il s'agit de montrer que

première partie

$$(x-y) + (a-b) = (x+a) - (y+b).$$

Montrons déjà que  $(0-a)+(0-b) = 0-(a+b)$

$$\begin{aligned} (0-a)+(0-b) &= (0-a)+(0-b) + 0 = (0-a)+(0-b) + ((a+b) - (a+b)) \\ &= (0-a)+(0-b) + ((a+b) + (0-(a+b))) \\ &= (a + (0-a)) + (b + (0-b)) + (0-(a+b)) \\ &= (a-a)+(b-b)+(0-(a+b)) = 0-(a+b) \end{aligned}$$

$$\begin{aligned} \text{alors } (x-y)+(a-b) &= (x+(0-y))+(a+(0-b)) \\ &= (x+a) + ((0-y)+(0-b)) = (x+a)+(0-(y+b)) \\ &= (x+a)-(y+b). \end{aligned}$$

On montre de même que  $PB(/,mult)$  en utilisant les axiomes MULTASS, MULTCOM, DIV/, IDIV/

R2: REGLE UTILISANT LA PROPRIETE (PB)

$$\begin{aligned} PB(f,g) \quad &|- \text{ ALL } g \circ (\text{trans } \circ [*f \circ \{x,a\}, *f \circ \{y,b\}]) \\ &= *f \circ [g \circ \{x,y\}, \text{ ALL } g \circ (\text{trans } \circ [a,b])]. \end{aligned}$$

La justification se fait simplement par induction sur la structure de la séquence "trans  $\circ [a,b]$ ".

Intuitivement, pour deux opérations "." et ":" telles que

$$(x.y):(a.b) = (x:a).(y:b), \text{ on a:}$$

$$\begin{aligned} ((x.a1).a2).a3)) : ((y.b1).b2).b3)) &= (((x.a1).a2) : ((y.b1).b2))) . (a3:b3) \\ &= (((x.a1):(y.b1)).(a2:b2)).(a3:b3) \\ &= (((x:y).(a1:b1)).(a2:b2)).(a3:b3) \end{aligned}$$

et c'est la même chose pour un nombre quelconque de ai.

EXEMPLES D'APPLICATION

a) Exemple 4 du chapitre I

première partie

Nous allons pouvoir montrer que les expressions (1) et (2) de l'exemple 4 du chapitre I sont A-égales

(1)  $+ o [ps o [ALL a o (iota o n), ALL b o (iota o n)]$   
 $, ps o [ALL c o (iota o n), ALL d o (iota o n)]]$   
 where def  $ps = last o ( ** o [0^-, ALL mult o trans])$

(2)  $last o (*f o [0^-, iota o n])$   
 where def  $f = + o [?1, + o [mult o [a o ?2, b o ?2]$   
 $, mult o [c o ?2, d o ?2]]$ .

En appliquant E2 et E1 à f, on a

$f =_A + o [?1, g o ?2]$  si  $g = + o [mult o [a, b], mult o [c, d]]$ .

On peut alors appliquer \*ALL à l'expression (2)

(2)  $=_A last o ( ** o [0^-, ALL g o (iota o n)])$ .

Avec ALLALL et FALL

$ALL g =_A ALL + o (trans$   
 $o [ALL mult o (trans o [ALL a, ALL b])$   
 $, ALL mult o (trans o [ALL c, ALL d]) ])$ .

Soient  $u = trans o [ALL a o (iota o n), ALL b o (iota o n)]$

et  $v = trans o [ALL c o (iota o n), ALL d o (iota o n)]$ ,

on peut appliquer E1 et E2 pour avoir

(2)  $=_A last o ( ** o [0^-, ALL + o (trans o [u, v]) ])$ .

Or nous savons que PB(+,+) et si l'on sait que 0 est élément neutre de "+" (avec l'axiome PLUS0), on peut appliquer R2 à l'expression (2).

première partie

On a donc

(2)  $=_A last o (ALL + o (trans$   
 $o [ ** o [0^-, ALL mult o u]$   
 $, ** o [0^-, ALL mult o v] ]))$ .

En appliquant alors E3 et E2, on a

(2)  $=_A last o (ALL + o (trans$   
 $o [ ( ** o [0^-, ALL mult]) o u$   
 $, ( ** o [0^-, ALL mult]) o v ]))$ .

Il suffit d'appliquer E1, LALL et LTRANS pour avoir

(2)  $=_A (1)$ .

La règle R2 permet de résoudre cet exemple qui nécessite une demande d'eureka dans le système de R. BURSTALL et J. DARLINGTON [Bur77].

b) Exemple 3 du chapitre I.

Nous allons pouvoir montrer que les expressions (1) et (4) de l'exemple 3 du chapitre I sont A-égales.

(1)  $last o (ALL / o (trans o [u, v]))$   
 where def  $u = *mult o [1^-, ALL 2^- o (iota o n)]$   
 and def  $v = *mult o [1^-, iota o n]$

(4)  $last o (*g o [1^-, iota o n])$   
 where def  $g = mult o [?1, / o [2^-, ?2]]$

On sait que PB(mult,/). On peut appliquer R2 à l'expression (1)

(1)  $=_A last o (*mult o [/ o [1^-, 1^-],$   
 $ALL / o (trans o [ALL 2^- o (iota o n), iota o n]) ])$ .

Maintenant on peut appliquer l'axiome IDIV/

Première partie

$$/ o [1^-, 1^-] =_A 1^-$$

et avec ALLID, E2 et FALL

$$\text{trans } o [ALL \ 2^- \ o \ (iota \ o \ n), iota \ o \ n] =_A ALL[2^-, id] \ o \ (iota \ o \ n)$$

alors

$$(1) =_A \text{last } o \ (*mult \ o \ [1^-, ALL \ / \ o \ (ALL[2^-, id] \ o \ (iota \ o \ n))]).$$

On peut alors appliquer ALLALL puis \*ALL

$$(1) =_A \text{last } o \ (*mult \ o \ [?1, (/ \ o \ [2^-, id]) \ o \ ?2]) \ o \ [1^-, iota \ o \ n]).$$

On applique alors E1, E2, E3, P1

$$(1) =_A \text{last } o \ (*mult \ o \ [?1, / \ o \ [2^-, ?2]) \ o \ [1^-, iota \ o \ n])$$

c'est-à-dire

$$(1) =_A (4).$$

#### 4.3. Troisième règle

PROPRIETE (PC)

La propriété met en jeu trois opérations que l'on peut noter ":", ":",  
 "..." et une fonction s (fonction dans les entiers).

$$(x.i):s(i) = (x:i)..s(i)$$

Ce qu'on formule fonctionnellement par

$$PC(g, f, h, s) \quad g \ o \ [f \ o \ [x, i], s \ o \ i] = h \ o \ [g \ o \ [x, i], s \ o \ i].$$

première partie

On peut remarquer que PC(f, f, f, s) est trivialement vérifiée quelles que soient f et s.

Si nous reprenons les fonctions "inv", "decal", "place" et "val" de l'exemple 10 du chapitre I, on a une propriété PR2:

$$\text{decal } o \ [inv \ o \ [t, i], + \ o \ [1^-, i]] = \text{decal } o \ [\text{decal } o \ [t, i], + \ o \ [1^-, i]]$$

qui exprime que si, dans le tableau t, l'on inverse les éléments de rang i et i+1 puis que l'on déplace l'élément i+2 en i+1, cela revient à faire deux décalages successifs.

PR2 exprime PC(decal, inv, decal, + o [1^-, id]).

On a aussi une propriété PR3:

$$\text{val } o \ [inv \ o \ [t, i], + \ o \ [1^-, i]] = \text{val } o \ [t, i]$$

qui exprime que la valeur en i+1 dans le tableau t modifié par l'inversion des éléments i et i+1 est la valeur initialement en i.

En utilisant l'axiome P3, l'on peut aussi écrire PR3 sous la forme

$$\text{val } o \ [inv \ o \ [t, i], + \ o \ [1^-, i]] =_A ?1 \ o \ [\text{val } o \ [t, i], + \ o \ [1^-, i]]$$

et l'on a PC(val, inv, ?1, + o [1^-, id]).

R3: UNE REGLE UTILISANT LA PROPRIETE (PC)

$$PC(g, f, h, s) \quad |- \quad g \ o \ [\text{last } o \ (*f \ o \ [x, iota \ o \ n]), s \ o \ n]$$

$$= \text{last } o \ (*h \ o \ [g \ o \ [x, 1^-], ALL \ s \ o \ (iota \ o \ n)]).$$

première partie

On justifie R3 avec une récurrence sur l'entier n.

Intuitivement, si  $PC(.,.,.,s)$ , on a

$$\begin{aligned}(x.i):s(i) &= (x.i)..s(i) \text{ donc} \\ ((x.1).2).3:s(3) &= (((x.1).2):3)..s(3) = (((x.1):2)..s(2))..s(3) \\ &= (((x:1)..s(1))..s(2))..s(3).\end{aligned}$$

APPLICATIONS AUX FONCTIONS inv, decal ET val DE L'EXEMPLE 10 DU CHAPITRE 1.

Nous avons vu que  $PC(val, inv, ?1, + o [1^-, id])$ , en appliquant R3, on a

$$\begin{aligned}&val o [last o (*inv o [t, iota o n]), + o [1^-, n]] \\ &=_{\mathcal{A}} last o (*?1 o [val o [t, 1^-], ALL(+ o [1^-, id]) o (iota o n)]])).\end{aligned}$$

Appelons v le deuxième membre, on peut simplifier v en appliquant LTL et

E1 qui permettent d'appliquer TL\*?1, on a

$$\begin{aligned}v &=_{\mathcal{A}} last o \\ & (ALL ?1 o (dist1 o [val o [t, 1^-], ALL(+ o [1^-, id]) o (iota o n)]))).\end{aligned}$$

On applique alors LALL, LDIST1 et P3

$$v =_{\mathcal{A}} val o [t, 1^-].$$

On a donc

$$\begin{aligned}(a) \quad &val o [last o (*inv o [t, iota o n]), + o [1^-, n]] \\ &=_{\mathcal{A}} val o [t, 1^-].\end{aligned}$$

La valeur de l'extrémité d'un tableau de longueur n obtenu par n inversions consécutives est la valeur du tableau original à l'indice 1.

#### CONSEQUENCE DE LA REGLE R3

première partie

$$\begin{aligned}PC(g, f, g, + o [1^-, id]) &|- g o [last o (*f o [x, iota o n]), + o [1^-, n]] \\ &= last o (*g o [x, iota o (+ o [1^-, n])]).\end{aligned}$$

Il suffit d'appliquer HIOTA, ALL+IOTA, TL\* et LTL au deuxième membre de R3 qui devient

$$last o (*g o [g o [x, 1^-], ALL(+ o [1^-, id]) o (iota o n)]).$$

REMARQUE

On peut appliquer cette conséquence pour une seule fonction f qui vérifie trivialement PC et on a

$$\begin{aligned}(R) \quad &f o [last o *f o [x, iota o n], + o [1^-, n]] \\ &=_{\mathcal{A}} last o (*f o [x, iota o (+ o [1^-, n])])\end{aligned}$$

qui est une définition de "last o \*f" appliquée à "iota".

APPLICATION AUX FONCTIONS "inv" ET "decal".

Nous avons vu que  $PC(decal, inv, decal, + o [1^-, id])$ .

En appliquant la conséquence de R3, on trouve

$$\begin{aligned}(b) \quad &decal o [last o (*inv o [t, iota o n]), + o [1^-, n]] \\ &=_{\mathcal{A}} last o (*decal o [t, iota o (+ o [1^-, n])]).\end{aligned}$$

Une suite de n inversions consécutives suivies d'un décalage de l'élément suivant revient à une suite de n+1 décalages.

On peut maintenant traiter l'exemple 10 du chapitre 1.

On veut montrer que les expressions (1) et (2) de l'exemple 10 du chapi-

première partie

tre I sont A-égales.

(1)  $\text{last } o (*\text{inv } o [t, \text{iota } o n])$

(2)  $\text{place } o [\text{val } o [t, l^-]$

$\text{, last } o (*\text{decal } o [\text{decal } o [t, l^-], \text{tl } o (\text{iota } o (+ o [l^-, n]))]$   
 $\text{, } + o [l^-, n] ]$ .

On utilisera une propriété PR1

PR1:  $\text{inv} = \text{place } o [\text{val}, \text{decal}, + o [l^-, \text{id}]]$ .

On applique (R) ci-dessus pour la fonction inv si l'on a les axiomes  
 IMOINS et MOINS qui nous permettent de montrer que

(M)  $+ o [l^-, - o [n, l^-]] =_A n$

et l'on trouve

(1)  $=_A \text{inv } o [\text{last } o (*\text{inv } o [t, \text{iota } o (- o [n, l^-])), n]$ .

On utilise PR1 pour inv à l'occurrence 21:

(1)  $=_A (\text{place } o [\text{val}, \text{decal}, + o [l^-, \text{id}]]$   
 $o [\text{last } o (*\text{inv } o [t, \text{iota } o (- o [n, l^-])), n] ]$ .

On peut alors appliquer E1, E2, E3 et P2

(1)  $=_A \text{place } o [\text{val } o [\text{last } o (*\text{inv } o [t, \text{iota } o (- o [n, l^-])), n]$   
 $\text{, decal } o [\text{last } o (*\text{inv } o [t, \text{iota } o (- o [n, l^-])), n]$   
 $\text{, } + o [l^-, n] ]$ .

En considérant (M), on applique alors (a) et (b) et on prouve

(1)  $=_A$  (2).

#### 4.4. Conclusion

##### Classes d'expressions A-égales

Notre ensemble d'axiomes A est composé pour pouvoir prouver que deux  
 expressions d'un programme itératif obtenues par deux raisonnements  
 structurés différents sont A-égales.

A contient pour cela:

- les axiomes sur les formes fonctionnelles de base,
- les axiomes qui correspondent pour les programmes itératifs aux règles  
 de composition des itérations,
- les axiomes qui correspondent aux règles de fusions d'itérations.

Les règles de fusions se codent par l'intermédiaire de la primitive  
 "trans" aussi est-il nécessaire d'avoir les axiomes sur cette primitive.

Les axiomes sur la primitive "dist1" (en particulier ALLDIST1 et TDDIST1)  
 sont utiles pour montrer les équivalences entre les expressions qui codent  
 deux programmes itératifs p1 et p2 qui diffèrent par la position d'un cal-  
 cul constant dans une itération. En effet, "dist1" sert à coupler une fonc-  
 tion (qui code un calcul constant d'une itération I) avec la séquence argu-  
 ment de I.

Les axiomes sur "iota" sont utiles pour supprimer une imbrication  
 superflue d'une itération dans une autre (axiome ALL\*IOTA).

Les règles d'inférence R1, R2 et R3 permettent de prouver des

équivalences de deux itérations I et J qui se déduisent de propriétés particulières aux fonctions itérées.

Une classe d'équivalence dans A comporte les expressions correspondant aux programmes itératifs équivalents par fusion, composition d'itération, places différentes des calculs constants d'une itération. Mais si on est un peu plus ambitieux, on aura besoin d'axiomes sur les primitives sur les booléens, les entiers, ..., mais aussi d'axiomes permettant de prouver la validité des conditions des axiomes conditionnels.

Mais notre but est de trouver dans une classe d'équivalence la meilleure expression qui, au sens défini dans le chapitre I, sera pour nous celle qui engendre le moins de séquences intermédiaires pour une exécution séquentielle. Nous allons pour cela utiliser A pour ériger un système de réécriture de termes de termes [Hue80a].

Il s'agit de donner une orientation aux axiomes de A qu'on appelle alors des règles. Alors, la relation de réduction, notée " $\Rightarrow$ ", ne permet plus que le remplacement d'une instance d'une partie gauche d'une règle par l'instance correspondante de sa partie droite.

La réécriture d'un terme t par la fermeture transitive de la relation de réduction, notée

$$" \Rightarrow^* "$$

peut permettre d'obtenir une forme canonique  $t \downarrow$  de ce terme dans la théorie équationnelle A. Autrement dit  $t \downarrow$  est le représentant de la classe d'équivalence des termes A-égaux à t.

première partie

Si de plus l'application du terme  $t \downarrow$  n'engendre pas de séquences intermédiaires superflues, alors  $t \downarrow$  doit être le résultat de la transformation de la classe des termes A-égaux à t. Le système de réécriture est le système de transformation solution de notre problème.

Pour cela il faut que le système de réécriture soit convergent, c'est à dire qu'il soit noethérien (le processus de réduction se termine toujours) et qu'il ait la propriété de CHURCH-ROSSER (si deux termes sont A-égaux, ils se réécrivent toujours en un même terme).

Si l'on a un ordre de réduction et un ensemble d'équations, la procédure de complétion de Knuth-Bendix [Knu70] produit, si elle se termine, un système de réécriture convergent. Nous voulons utiliser l'implémentation de cette procédure proposée par le logiciel REVE ([Les83] et [For84]).

Mais avec l'ensemble d'axiomes A, c'est, à notre connaissance, une des premières fois que l'on essaie d'appliquer la procédure de Knuth-Bendix à un aussi gros ensemble d'axiomes. C'est aussi la première fois que la procédure de Knuth-Bendix est utilisée de façon systématique pour produire des outils de transformation de programmes (la procédure de Knuth-Bendix est utilisée par [Che83], [Cho85] et [Cos85]). Nous allons relater notre expérience au chapitre III.

## 5. Bibliographie

Nous avons emprunté la terminologie sur les FP-termes à G. HUET [Hue80a]. Pour la présentation, nous nous sommes aussi servie des thèses de J. M. HULLOT [Hu180] et de C. et H. KIRCHNER [Kir82].

première partie

Pour les théories intégrant les équations conditionnelles, nous nous appuyons essentiellement sur le travail de J. L. REMY [Rem82].

La majeure partie de l'algèbre des FP-termes est décrite par J. BACKUS [Bac78]. Certains axiomes ont été auparavant cités par Z. MANNA [Man73]. Nous empruntons aussi des axiomes sur les conditionnelles décrits par S. BLOOM et R. TINDELL [Blo83].

Le chapitre III va essentiellement être consacré à l'application du système REVE [Les83] sur l'ensemble des axiomes A de la FP-Algèbre que nous venons d'introduire dans ce chapitre II.

## 1. Systèmes de réécriture de termes

### 1.1. définitions

Un système de réécriture de termes est un ensemble fini d'équations orientées. Une équation " $g = d$ " orientée est une règle de réécriture que l'on note " $g \Rightarrow d$ ". On doit avoir  $V(d) \subseteq V(g)$ .

A un système de réécriture, on associe une relation de réduction, que l'on note " $\Rightarrow$ ", telle que:

$t_1 \Rightarrow t_2$  si et seulement si il existe une règle  $g \Rightarrow d$ , une occurrence  $u$  de  $D(t_1)$  et une substitution  $\sigma$  telle que

$$\sigma(g) = t_1/u \text{ et que } t_2 = t_1[u \leftarrow \sigma(d)].$$

On dit alors que  $t_1$  se réécrit (ou se réduit) en  $t_2$ .

### EXEMPLE

Le système de réécriture est

- E1:  $(f \circ g) \circ h \Rightarrow f \circ (g \circ h)$ ,
- E2:  $[f, g] \circ h \Rightarrow [f \circ h, g \circ h]$ ,
- FALL:  $\text{trans} \circ [\text{ALL } f, \text{ALL } g] \Rightarrow \text{ALL } [f, g]$ ,
- ALLALL:  $\text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$  }

Le terme

$$t = (\text{ALL mult} \circ (\text{trans} \circ [\text{ALL } a, \text{ALL } b])) \circ (\text{iota} \circ n)$$

se réécrit en  $t_1$  à l'occurrence 12 avec la règle FALL

$$t_1 = (\text{ALL mult o ALL [a,b]}) \text{ o } (\text{iota o n})$$

Mais il se réduit aussi à l'occurrence  $\epsilon$  avec la règle E1 en

$$t_2 = \text{ALL mult o } ((\text{trans o [ALL a, ALL b]}) \text{ o } (\text{iota o n})).$$

" $\Rightarrow^*$ " désigne la fermeture transitive de la relation " $\Rightarrow$ ".

Un terme  $t$  est dit en forme normale (ou irréductible) si et seulement si il n'existe pas de terme  $t'$  tel que  $t \Rightarrow^* t'$ .

Si  $t$  et  $t'$  sont tels que  $t'$  est irréductible et  $t \Rightarrow^* t'$ ,  $t'$  est une forme normale de  $t$ .

#### EXEMPLE

Dans l'exemple précédent,  $t_1$  et  $t_2$  ne sont pas irréductibles.

$t_1$  se réécrit à l'occurrence 1 avec la règle ALLALL en

$$t_1' = \text{ALL}(\text{mult o [a,b]}) \text{ o } (\text{iota o n}) \text{ qui est irréductible.}$$

Mais il se réduit aussi à l'occurrence  $\epsilon$  avec E1 en

$$t_1'' = \text{ALL mult o } (\text{ALL[a,b] o } (\text{iota o n})) \text{ qui est irréductible.}$$

Quant à  $t_2$ , il se réduit à l'occurrence 21 avec FALL en

$$t_2' = \text{ALL mult o } (\text{ALL [a,b] o } (\text{iota o n})) \text{ qui est irréductible.}$$

Mais il se réduit aussi à l'occurrence 2 avec E1 en

$$\text{ALL mult o } (\text{trans o } ((\text{ALL a, ALL b}) \text{ o } (\text{iota o n})))$$

première partie

puis avec E2 à l'occurrence 22 en

$$t_2'' = \text{ALL mult o } (\text{trans o } [\text{ALL a o } (\text{iota o n}), \text{ALL b o } (\text{iota o n})])$$

qui est irréductible.

$t_1'$ ,  $t_1''$ ,  $t_2'$  et  $t_2''$  sont des formes normales de  $t$ .

#### Procédure de décision de la A-égalité

Soit  $A$  un ensemble d'équations orientées en un système de réécriture de termes  $R$ . Si on veut prouver que deux termes  $t_1$  et  $t_2$  sont A-égaux, on peut chercher une forme normale  $t_1 \downarrow_R$  de  $t_1$  et une forme normale  $t_2 \downarrow_R$  de  $t_2$ . Si  $t_1 \downarrow_R$  et  $t_2 \downarrow_R$  sont identiques, on peut affirmer que  $t_1$  et  $t_2$  sont A-égaux.

Cette méthode peut devenir une procédure de décision de la A-égalité à condition de pouvoir répondre aux deux questions suivantes

a) Le processus de réécriture d'un terme se termine-t-il dans tous les cas?

b) Un terme  $t$  a-t-il toujours une forme normale  $t \downarrow_R$  unique?

#### 1.2. Terminaison d'un système de réécriture

Un système de réécriture est noethérien (ou à terminaison finie) si et seulement si pour tout terme  $t$ , il n'existe pas de dérivation infinie

$$t = t_0 \Rightarrow t_1 \Rightarrow \dots$$

Quand un système est noethérien, le processus de réécriture se termine toujours. Tout terme  $a$  au moins une forme normale.

première partie

Le problème de la terminaison d'un système de réécriture est indécidable. Mais, il existe des méthodes pour prouver la terminaison d'un système de réécriture.

### Ordre de réduction

Un ordre de réduction, noté ">" est un ordre qui vérifie la propriété de compatibilité, c'est à dire que pour tout symbole fonctionnel "op" (d'arité n) et tout n-uple de termes " $u_1, \dots, \_, \dots, u_n$ " (" $\_$ " indique la place de t ou t' dans le n-uple), on a

$$t > t' \Rightarrow \text{op}(u_1, \dots, t, \dots, u_n) > \text{op}(u_1, \dots, t', \dots, u_n)$$

### Méthode de LANKFORD

#### Théorème de MANNA&NESS [Man70]

Un système de réécriture de termes  $\{g_i \Rightarrow d_i\}$  sur un ensemble de termes est noethérien si il existe un ordre de réduction noethérien ">" tel que pour toute règle d'étiquette i et pour toute substitution  $\sigma$ , on ait

$$\sigma(g_i) > \sigma(d_i).$$

#### La méthode de LANKFORD

est fondée sur l'application du théorème ci-dessus [Lan79].

On utilise une interprétation I des termes sur un domaine bien fondé tel que, pour tout symbole fonctionnel op, I(op) soit une fonction monotone croissante pour chacun de ses arguments.

Pour prouver la terminaison du système  $\{g_i \Rightarrow d_i\}$ , il suffit alors de

première partie

vérifier que, pour tout i et pour toute substitution  $\sigma$ ,

$$I(\sigma(g_i)) > I(\sigma(d_i))$$

### EXEMPLE

Avec l'interprétation:

$$I(o) = \lambda xy. x(y+1), \quad I([\_]) = \lambda xy. (x+y+1)$$

sur les entiers supérieurs à 1 et pour le système de réécriture E1E2:

$$\{[f, g] \circ h \rightarrow [f \circ h, g \circ h], (f \circ g) \circ h \rightarrow f \circ (g \circ h)\}$$

On a pour la première règle

$$\begin{aligned} I([f, g] \circ h) &= I([f, g])(I(h) + 1) = (I(f) + I(g) + 1)I(h) + (I(f) + I(g) + 1) \\ &= I(f)I(h) + I(g)I(h) + I(h) + I(f) + I(g) + 1 \end{aligned}$$

et

$$\begin{aligned} I([f \circ h, g \circ h]) &= I(f \circ h) + I(g \circ h) + 1 = I(f)(I(h) + 1) + I(g)(I(h) + 1) + 1 \\ &= I(f)I(h) + I(f) + I(g)I(h) + I(g) + 1 \end{aligned}$$

puisque  $I(h) > 1$ , on a donc:

$$I([f, g] \circ h) > I([f \circ h, g \circ h]).$$

Le raisonnement reste le même pour toute substitution  $\sigma$ .

Pour la deuxième règle:

$$\begin{aligned} I((f \circ g) \circ h) &= I(f)(I(g) + 1)(I(h) + 1) \\ &= I(f)I(g)I(h) + I(f)I(g) + I(f)I(h) + I(f) \end{aligned}$$

$$\begin{aligned} I(f \circ (g \circ h)) &= I(f)(I(g)(I(h) + 1) + 1) \\ &= I(f)I(g)I(h) + I(f)I(g) + I(f) \end{aligned}$$

Puisque  $I(f)I(g) > 1$ , on a bien:

$$I((f \circ g) \circ h) > I(f \circ (g \circ h)),$$

première partie

ce qui reste vrai pour toute substitution  $\sigma$ .

Le système E1E2 est noethérien.

### Méthode de DERSHOWITZ

La propriété de noethérianité exigée pour les ordres de réduction est évitée par l'utilisation d'ordres de simplification.

#### Ordre de simplification

Un ordre partiel sur les termes est un ordre de simplification si, pour tous termes  $t$  et tous les couples  $t_2$  et  $t_3$ , il vérifie les deux propriétés de

a) sous-terme:  $t < \text{op}(u_1, \dots, t, \dots, u_n)$  pour tout  $t$ .

b) compatibilité:

$$t_1 < t_2 \Rightarrow \text{op}(u_1, \dots, t_1, \dots, u_n) < \text{op}(u_1, \dots, t_2, \dots, u_n)$$

#### Théorème de Dershowitz ([Der82a],[DER82b])

Un système de réécriture de termes  $\{g_i \Rightarrow d_i\}$  se termine s'il existe un ordre de simplification tel que, pour toute règle d'étiquette  $i$ , et pour toute substitution  $\sigma$ , on ait  $\sigma(g_i) > \sigma(d_i)$ .

Il suffit donc d'avoir des ordres de simplification.

#### Des ordres de simplifications

L'ordre récursif sur les chemins ("recursive path ordering" (ou RPO)) est proposé par PLAISTED, DERSHOWITZ et KAMIN&LEVY ([Pla78],[Der82a]). L'ordre

première partie

récursif de décomposition ("recursive decomposition ordering" (ou RDO)) est proposé par JOUANNAUD, LESCANNE et REINIG ([Jou81b],[Jou82b],[Les81],[Les84a]). Ils sont tous deux fondés sur une extension de la notion de précédence qui est un ordre sur les symboles.

Nous présentons maintenant ces deux ordres.

### L'ordre RPO

Le RPO réclame un statut pour chaque opérateur. Ce statut peut être multi-ensemble ("multiset") ou lexicographique de gauche à droite (LR) ou de droite à gauche (RL).

Si deux termes ont, au sommet, le même symbole avec un statut (LR), les sous termes gauche sont considérés en premier et le terme le plus grand est celui qui a le premier le plus grand sous-terme gauche (comme pour l'ordre lexicographique habituel).

Par exemple, si on donne au symbole "o" le statut LR, on a

$$(f \circ g) \circ h > f \circ (g \circ h)$$

Le RPO compare les termes en examinant d'abord les symboles de tête et en comparant récursivement les termes et ses sous-termes directs selon une stratégie déterminée par le résultat de la comparaison.

#### Définition du RPO

Soient deux termes

$$t_1 = \text{op1}(u_1, \dots, u_n) \text{ et } t_2 = \text{op2}(v_1, \dots, v_m).$$

première partie

On dit que  $t_1 < t_2$  si

a)  $op1 < op2$  et pour tout  $i$ ,  $u_i < t_2$

b)  $op1 = op2$ , pour tout  $i$ ,  $u_i < t_2$  et  $u_1, u_2, \dots, u_n < v_1, v_2, \dots, v_n$

(si le statut est multiset "<<" est l'ordre de comparaison des multi-ensembles et si le statut est LR (ou RL), "<<" est l'ordre de comparaison lexicographique de gauche à droite (ou de droite à gauche))

c) il existe un  $i$  tel que  $t_1 < v_i$  ou  $t_1 = v_i$

#### EXEMPLES

1)  $f \circ (g \circ h) < (f \circ g) \circ h$

si le symbole "o" a le statut LR, car alors

$$f < (f \circ g).$$

2)  $[f \circ h, g \circ h] < [f, g] \circ h$

avec la précedence  $\circ > [ ]$ , puisqu'en ce cas

$$f \circ h < [f, g] \circ h \text{ et } g \circ h < [f, g] \circ h.$$

#### L'ordre RDO

Le RDO commence par rechercher dans le terme une décomposition. Cette décomposition permet de trouver quels sont les symboles qui, dans le terme, jouent le rôle de leaders en fonction de leur place dans le terme et de leur précedence. Pour comparer des termes, il compare leurs décompositions.

première partie

#### Introduction

Une définition complète du RDO est assez longue. Nous empruntons à [Les84a] une introduction informelle à l'aide de deux petits exemples types:

exemple 1:

Si " $i > *$ "

alors " $i(x * y) > *(i(x), i(y))$ " car

"le symbole le plus lourd descend tandis que le plus léger monte".

exemple 2:

Si " $\& > !$ "

alors " $\neg(\&(x, y)) > !(\neg x, \neg y)$ " car

" $\neg$ " descend et " $\&$ " est remplacé par un symbole plus petit.

exemple 3:

Pour comparer " $(f \circ g) \circ h$ " et " $f \circ (g \circ h)$ ", il faut associer un statut à LR ou RL à "o".

exemple 4:

De plus, le RDO teste que les autres termes "ne sont pas trop gros":

Si " $/$ " a un statut LR et que  $b$  et " $/$ " sont équivalents alors

$$x/(y/z) > (x/b(z))/y$$

première partie

- car - y est plus petit que "y/z",
- "x/b(z)" n'est pas "trop gros" (il est plus petit que "x/(y/z)")
- et x est plus petit que "x/(y/z)".

### Propriétés

#### 1) Comparaison des ordres RDO et RPO:

En général, le RPO et le RDO donnent le même résultat. Mais, dans le cas où un symbole de tête a un statut multi-ensemble, il existe des termes qui ne sont pas comparables avec le RPO et qui sont comparables avec le RDO.

#### EXEMPLE

Avec le RPO, si " $o[_]>trans$ ", " $trans\ o\ [ALL\ f\ o\ h, ALL\ g\ o\ h]$ " et " $ALL[f,g]\ o\ h$ " ne sont pas comparables, puisque " $[ALL\ f\ o\ h, ALL\ g\ o\ h]$ " ne peut être inférieur ni à " $ALL[f,g]$ " ni à h.

Mais le RDO peut répondre que le premier est inférieur puisque "ALL" monte tandis que " $[_]$ " descend.

#### 2) Une propriété très importante du RDO est son "incrémentalité".

Le RDO permet de faire des suggestions pour étendre la précédence quand celle-ci se révèle insuffisante pour comparer deux termes.

Ces suggestions proviennent des leaders des deux termes et peuvent prendre la forme d'un ensemble ordonné de paires de symboles. Dans de nombreux cas, l'extension de la précédence est décidable. Le RDO peut donc permettre, dans certains cas, de trouver la précédence.

L'ordre RPO n'est défini qu'accompagné de la précédence et des statuts des opérateurs. L'ordre RDO se définit "tout seul" ou, tout au moins, d'une manière incrémentale.

### 1.3. Confluence

Quand un système de réécriture de termes est confluent, on a aussi l'unicité de la forme normale d'un terme.

Le système donné en exemple au paragraphe 1.1 n'est manifestement pas confluent puisque nous avons exhibé plusieurs formes normales pour un même terme t.

#### Définition de la confluence

Un système de réécriture est confluent si et seulement si, pour tous les termes t,  $t_1$  et  $t_2$  tels que

$$t \Rightarrow^* t_1 \text{ et } t \Rightarrow^* t_2,$$

il existe un terme  $t'$  tel que

$$t_1 \Rightarrow^* t' \text{ et } t_2 \Rightarrow^* t'.$$

#### Confluence et propriété de CHURCH ROSSER

La propriété de confluence est équivalente à la propriété de Church-Rosser:

Si deux termes  $t_1$  et  $t_2$  sont A-égaux, il existe un terme  $t'$  tel que

$$t_1 \Rightarrow^* t' \text{ et } t_2 \Rightarrow^* t'$$

### Convergence

Un système noethérien et confluent implique l'existence et l'unicité d'une forme normale pour tout terme  $t$ . Nous disons alors que le système de réécriture est convergent.

### Confluence locale

Pour rendre la propriété de confluence décidable, il est intéressant de pouvoir la localiser.

Un système de réécriture est localement confluent si et seulement si, pour tous termes  $t$ ,  $t_1$  et  $t_2$  tels que

$$t \Rightarrow t_1 \text{ et } t \Rightarrow t_2,$$

il existe un terme  $t'$  tel que

$$t_1 \Rightarrow^* t' \text{ et } t_2 \Rightarrow^* t'.$$

Or quand un système noethérien est localement confluent, il est confluent.

Autrement dit, un système de réécriture de termes noethérien et localement confluent est convergent.

### 1.4. Algorithme de Knuth-Bendix

Une procédure de Knuth-Bendix essaie de transformer un ensemble  $A$  d'équations en un système de réécriture convergent "équivalent".

"équivalent" signifie ici que deux termes  $A$ -égaux  $t_1$  et  $t_2$  ont la même

première partie

forme normale pour le système de réécriture résultat de la procédure de Knuth-Bendix appliqué à  $A$ .

### Filtrage et unification

Une substitution  $\sigma$  est un filtre de  $t_1$  vers  $t_2$  si et seulement si  $\sigma(t_1) = t_2$ .

L'ensemble des substitutions peut être ordonné:

$\sigma_1 < \sigma_2$  si et seulement si

il existe une substitution  $\sigma$  telle que  $\sigma\sigma_1 = \sigma_2$ .

Une substitution  $\sigma$  est un unificateur des deux termes  $t_1$  et  $t_2$  si et seulement si  $\sigma(t_1) = \sigma(t_2)$ .

L'unificateur est minimal s'il n'existe pas un autre unificateur  $\sigma'$  de  $t_1$  et  $t_2$  tel que  $\sigma' < \sigma$ .

### Paires critiques

Soient deux règles  $g_1 \Rightarrow d_1$  et  $g_2 \Rightarrow d_2$ , et supposons qu'il existe un sous-terme non variable de  $g_1$  à l'occurrence  $u$  unifiable avec  $g_2$  avec un unificateur minimal  $\sigma$ , alors la paire

$$(\sigma(g_1[u \leftarrow d_2]), \sigma(d_1))$$

est une paire critique obtenue par superposition de  $g_2 \Rightarrow d_2$  sur  $g_1 \Rightarrow d_1$ .

### EXEMPLE

La règle  $\text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$

première partie

se superpose sur la règle

$$(f \circ g) \circ h \Rightarrow f \circ (g \circ h) \text{ à l'occurrence 1.}$$

La paire critique est

$$( \text{ALL}(f \circ g) \circ h , \text{ALL } f \circ (\text{ALL } g \circ h) ).$$

Remarquons que si  $(t_1, t_2)$  est une paire critique, les deux termes  $t_1$  et  $t_2$  sont A-égaux.

#### Théorème de KNUTH-BENDIX

Un système de réécriture de termes est localement confluent si et seulement si, pour toute paire critique  $(t_1, t_2)$ , il existe un terme  $t'$  tel que

$$t_1 \Rightarrow^* t' \text{ et } t_2 \Rightarrow^* t'$$

Dans le cas où le système est noethérien, ce théorème fournit une procédure de décision de la confluence du système (donc de sa convergence).

La procédure de complétion de Knuth-Bendix fait plus: elle complète le système de réécriture pour le rendre localement confluent.

#### La procédure de KNUTH-BENDIX

Etant donné un système d'équations et un ordre de réduction, la procédure

1) oriente les équations en règles de réécriture, ce qui teste la terminaison du système.

2) recherche les paires critiques et regarde si elles sont convergentes, ce

première partie

qui teste la confluence locale.

3) complète l'ensemble de règles dans le cas où elle trouve une paire critique  $(t_1, t_2)$  non convergente, de la manière suivante:

Soient  $t_1 \downarrow_R$  et  $t_2 \downarrow_R$  des formes normales de  $t_1$  et  $t_2$  pour le système de réécriture  $R$ , la procédure ajoute alors l'équation  $t_1 \downarrow_R = t_2 \downarrow_R$  à l'ensemble.

Ainsi la procédure peut engendrer un système de réécriture convergent équivalent à l'ensemble d'équations fourni au départ.

#### REMARQUE

La procédure maintient les règles sous forme interréduite (les membres sont en forme normale pour le système de règles en cours de génération).

Donc

a) des règles peuvent disparaître au cours de la complétion.

b) le membre gauche d'une règle peut alors se trouver réduit ce qui oblige à réexaminer son orientation. Remarquons que c'est inutile quand la procédure est amenée à réduire le membre droit.

#### Application de la procédure de complétion

Deux sortes de problèmes peuvent survenir:

a) La procédure peut "diverger"

La procédure peut en effet être amenée à ajouter indéfiniment de nouvelles règles pour compléter le système.

Il se peut alors qu'une conséquence équationnelle non encore trouvée par

première partie

la procédure fasse disparaître les paires critiques responsables de la divergence. L'ordre d'introduction des nouvelles règles peut avoir une grande importance.

Il se peut aussi que la divergence disparaisse si on oriente une des règles différemment.

b) Les ordres de réduction permettant à la procédure de vérifier la terminaison finie peuvent être insuffisants.

Il se peut alors que l'équation non orientable puisse disparaître avec l'introduction de nouvelles règles.

Il se peut aussi qu'un autre ordre puisse convenir :

- soit parce qu'il oriente différemment les équations et que l'équation non orientable n'apparaisse plus,

- soit parce qu'il est capable d'orienter cette équation.

Il faut donc épuiser toutes les combinaisons possibles pour orienter les règles: une orientation est "possible" si elle est compatible avec les ordres implémentés dans la procédure.

Et encore, tous les ordres possibles ne sont pas implémentés. On doit donc pouvoir orienter une règle "manuellement" (c'est à dire sans garantie d'un ordre implémenté avec la procédure). Dans ce cas, la procédure ne peut garantir que la propriété de confluence locale.

Mais dans le cas d'un axiome permutatif comme " $x + y = y + x$ ", on est sûr qu'aucun ordre bien fondé ne peut comparer les deux membres.

première partie

## Conclusion

Quand la procédure se termine sans qu'aucune règle n'ait été orientée "manuellement": on a un système de réécriture convergent.

Quand la procédure se termine et qu'une règle a été orientée manuellement: on a un système de réécriture localement confluent. Dans ce cas, on peut prouver "manuellement" la terminaison par une méthode non implantée.

Quand la procédure diverge (c'est à dire boucle en engendrant des règles nouvelles) ou dans le cas où elle échoue à orienter une règle, c'est l'intuition de l'utilisateur qui peut essayer de tirer parti de la connaissance qu'il a de la théorie pour savoir que faire :

- revenir en arrière et orienter différemment une des règles?
- ajouter une conséquence équationnelle à l'ensemble d'équations?

Plus l'ensemble d'axiomes est important, plus cela devient difficile. La procédure de Knuth-Bendix implémentée dans le système REVE offre des services dont nous allons pouvoir tirer parti.

## 2. Appliquer REVE sur les axiomes de la FP-algèbre

Nous utilisons la procédure de Knuth-Bendix implémentée dans le système REVE.

### 2.1. Les particularités de REVE

L'utilisateur donne l'ensemble des équations

première partie

Implicitement, REVE considère que la présentation des équations présuppose une orientation de gauche à droite pour les règles correspondantes.

L'option "FREE" laisse le choix à REVE qui considère alors les deux sens comme également possibles (c'est forcément toujours le cas des équations ajoutées par la procédure de Knuth-Bendix).

#### Dans REVE, c'est l'ordre RDO qui est implémenté

REVE utilise donc la propriété d'incrémentalité du RDO et, dans la plupart des cas, REVE est capable de se débrouiller seul pour trouver la précédence.

Il ne requiert l'aide de l'utilisateur que si pour une équation  $t_1 = t_2$ , il ne sait pas prouver que  $t_1 > t_2$  (ou  $t_2 > t_1$ ) avec l'ordre RDO.

Pour les nouvelles équations (ou avec l'option FREE), il demande à l'utilisateur si l'orientation choisie lui convient. Dans le cas où il parvient à comparer les termes en déterminant la précédence, il présente la règle à l'utilisateur.

#### L'utilisateur intervient quand REVE ne trouve pas la précédence

Alors on peut choisir:

a) d'orienter l'équation dans le sens inverse de sa présentation.

b) d'étendre la précédence:

quand il le peut, (c'est presque toujours le cas) REVE guide le choix en suggérant des inégalités entre les symboles fonctionnels qui sont

première partie

susceptibles de l'aider à débloquer la situation.

Dans tous les cas REVE vérifie que l'on n'introduit pas d'inconsistance dans la précédence.

c) affecter un statut à un symbole:

Étendre la précédence n'est pas toujours suffisant. Il faut parfois donner un statut LR ou RL à un opérateur (c'est ce que l'on doit faire pour un opérateur associatif comme "o").

REVE vérifie la consistance avec l'ordre antérieur.

d) ajouter un nouveau symbole:

Dans le cas où il n'y a pas d'espoir de comparaison ni dans un sens ni dans l'autre avec le RDO et que  $V(t_2) \subseteq V(t_1)$  (ou vice versa), on peut aussi espérer pouvoir orienter en ajoutant un nouvel opérateur d'arité égale à la taille de l'intersection de  $V(t_1)$  et  $V(t_2)$ .

e) recommencer la procédure en un point quelconque de son déroulement en essayant une autre combinaison pour orienter les règles (avec la commande UNDO).

f) demander une orientation "manuelle" dans le cas où le RDO s'avoue incapable d'orienter une règle (qui est supposée orientable avec un ordre bien fondé et n'est donc pas permutative).

g) interrompre le déroulement de la procédure:

Cette interruption est utile dans le cas où l'on veut examiner la situation pour "rejeter" une équation parce que:

première partie

- c'est une nouvelle équation permutative (n'est pas orientable avec un ordre bien fondé).

- on s'aperçoit que REVE diverge en décelant une régularité dans la production des nouvelles équations. On veut alors "jeter" les paires critiques ou les équations qu'on estime à l'origine du "bouclage".

Dans ces deux cas, continuer le processus peut permettre de voir un ensemble de conséquences équationnelles de la théorie qui peut fournir des renseignements intéressants, voire de trouver l'équation à ajouter pour débloquer la situation.

Si on a rejeté des équations, REVE les considère de nouveau à la fin de la complétion.

Si on a rejeté des règles, et que REVE se termine, le système de réécriture obtenu n'est pas équivalent à la théorie initiale.

### Conclusion

En fait, nous proposons une utilisation originale de la procédure de Knuth-Bendix. Cette utilisation est beaucoup plus expérimentale que les utilisations classiques.

En effet, puisque la procédure ne peut se terminer, nous tentons, par une approche expérimentale, de tirer parti des résultats partiels qu'elle fournit et d'en inférer des conséquences. Elle devient, par conséquent, une méthode de "complétion assistée par ordinateur", d'automatique qu'elle était à l'origine. Pour cela nous utilisons les nombreux outils présents dans le laboratoire de réécriture de REVE.

Cette vue peut paraître restrictive, mais c'est la seule possible pour l'attaque du difficile problème de la programmation fonctionnelle, comme en témoignent les expériences que des collègues nous ont relatées en France et à l'étranger.

Ce chapitre doit donc être considéré comme une analyse des expériences que nous avons mené sur les transformations de programmes itératifs codés en FP.

### 2.2. Utiliser REVE sur l'ensemble d'équations défini au chapitre II

L'ensemble d'équations de la FP-algèbre que nous avons défini au chapitre II ne peut pas, à priori, être une donnée de REVE. Nous allons tenter de trouver une solution pour chacun des problèmes posés.

#### 1) La théorie a un nombre infini d'équations

A cause de l'arité multiple de "[...]" et du nombre infini des sélecteurs, la théorie a un nombre infini d'équations. Elle ne peut donc, sans introduire de nouveaux opérateurs, être équivalente à un système de réécriture qui a un nombre fini de règles.

Nous résoudrons complètement ce problème dans la deuxième partie. Dans une première approximation, nous fixons l'arité de "[\_]" à 2 et nous ne gardons que les sélecteurs "?1" et "?2". Cela nous permet d'étudier la théorie.

Cette restriction ne nous empêche pas d'obtenir un système de réécriture utilisable. En effet "[...]" sert à construire les arguments des

fonctions. Notons que les primitives sur les séquences (sauf "trans") ont une arité inférieure ou égale à 2, que les fonctions construites par " $\ast f$ ", " $WH(p,f)$ " sont d'arité 2 et que les fonctions construites par " $ALL f$ " sont d'arité 1. En limitant aussi l'arité de "trans" à 2, l'approximation demandée n'est pas ridicule (les expressions des exemples 1 à 7 du chapitre I peuvent être réduits dans le cadre de cette approximation).

## 2) La théorie contient des axiomes permutatifs

Il y a les axiomes de commutativité des primitives comme " $+$ " (PLUSCOM), " $\text{mult}$ " (PLUSMULT). Mais nous avons aussi un axiome permutatif sur " $\_ \rightarrow \_ ; \_$ " qui est l'axiome C

$$C: p \rightarrow (q \rightarrow f; g); (q \rightarrow x; y) = q \rightarrow (p \rightarrow f; x); (p \rightarrow g; y)$$

Une version (en cours d'implémentation par C. et H. KIRCHNER ([Kir84],[Kir85a]) ainsi que J. MZALI au CRIN et K. YELICK au MIT) de REVE permet d'obtenir des systèmes de réécriture modulo une théorie équationnelle qui peut ainsi inclure des axiomes permutatifs. Faute d'avoir encore pu utiliser ce nouveau système dans nos expériences, nous écartons les axiomes permutatifs pour le moment. Nous les considérerons dans la deuxième partie.

## 3) La théorie n'est pas purement équationnelle

### a) présentation du problème

Des axiomes de la FP-algèbre sont des équations accompagnées de préconditions d'application (axiomes conditionnels).

### EXEMPLES

première partie

$$P3: \text{DEF } o \ g :: ?1 \ o \ [f,g] = g$$

$$LTL: \text{DEF } o \ (\text{last } o \ t1) :: \text{last } o \ t1 = \text{last}$$

$$TLAPP: \text{DEF } o \ (t1 \ o \ a) :: t1 \ o \ (\text{append } o \ [a,b]) = \\ \text{append } o \ [t1 \ o \ a, b]$$

$$TLWH: \text{DEF } o \ [x, \text{hd } o \ a] :: t1 \ o \ (WH(p,f) \ o \ [x,a]) = \\ WH(p,f) \ o \ [f \ o \ [x, \text{hd } o \ a], t1 \ o \ a]$$

Nous avons vu (au chapitre II paragraphe 2.4 b) comment intégrer ces équations conditionnelles dans notre théorie équationnelle.

Nous avons en particulier remarqué (au chapitre II paragraphe 2.4 c) que l'application d'une équation conditionnelle c'est à dire son utilisation pour remplacer dans une expression dite "correcte" un sous-terme par son équivalent n'entraînait pas toujours la nécessité de vérifier la validité de la précondition.

Rappelons que les deux conditions exigées portent:

1) la première sur l'équation conditionnelle qui doit être de la forme

$$\text{DEF } o \ p :: t_1 = t_2$$

et signifie que " $t_1 \sqsubseteq t_2$ " ( $t_1$  est "moins défini que") ou/et " $t_2 \sqsubseteq t_1$ ".

La plupart des axiomes (par exemple P3, LTL et TLAPP) ont des préconditions convenables. Celles qui ne conviennent pas (par exemple celle de l'axiome TLWH) ont toujours été soulignées.

2) la deuxième sur l'utilisation de l'équation conditionnelle.

Quand  $t_1 \sqsubseteq t_2$ , l'utilisation de l'équation peut se faire sans vérifier la précondition uniquement pour remplacer dans une expression e correcte un

première partie

terme  $\sigma(t_1)$  par un terme  $\sigma(t_2)$ .

Notons que, la vérification de la précondition "DEF o  $\sigma(t)$ " doit être faite quand on remplace  $\sigma(t_2)$  par  $\sigma(t_1)$ .

#### EXEMPLES

Les axiomes suivants peuvent être appliqués sans vérifier la validité de leur précondition

P3: pour remplacement " $?1$  o  $[f,g]$ " par  $f$  et non l'inverse.

LTL: pour remplacer "last o  $t1$ " par "last" et non l'inverse.

TLAPP: pour remplacer "append o  $[t1$  o  $a,b]$ " par " $t1$  o (append o  $[a,b])$ " et non l'inverse.

Une version (en cours de définition et d'implémentation par J. L. REMY et A. ZHANG [Rem84]) de REVE prévoit les systèmes de réécriture conditionnels et nous regarderons dans la deuxième partie comment nous pourrions envisager d'utiliser cette version pour nos axiomes conditionnels.

#### b) Solution de remplacement envisagée

REVE dans l'état où nous l'avons utilisé ne manipule pas les axiomes conditionnels. En effet, le logiciel REVEUR 4 de J.L. REMY et A. ZHANG [Rem84], bien que prometteur est trop rudimentaire dans le type des conditions qu'il traite. Il ne pouvait pas être utilisé ici. Aussi nous avons tenté de simuler ou d'éviter les préconditions par une stratégie d'application des règles.

Si la réduction par la règle revient à une utilisation de l'équation

première partie

correspondante qui ne nécessite pas de vérification de la précondition, la précondition n'a pas non plus à être testée pour la réduction d'une expression correcte. Il suffit pour cela que, la précondition étant "DEF o  $t$ ",  $t$  soit un sous-terme bien placé dans le membre gauche de la règle. Cette constatation permet, quand l'expression réduite est correcte d'éviter la vérification de la précondition. Au niveau de la procédure de Knuth-Bendix, il faudrait en tenir compte pour orienter les règles. Il faudrait aussi essayer de trouver les préconditions convenables pour les nouvelles règles. Il ne faut pas non plus inclure les axiomes sur " $!$ " qui s'expriment, par définition, avec des expressions qui ne sont pas correctes.

Nous n'avons pas voulu intervenir dans REVE pour implanter cette stratégie car ce n'était ni dans notre objectif, ni de notre compétence. Nous envisagerons une solution à ce problème dans la deuxième partie.

Nous avons pensé qu'il était beaucoup plus judicieux pour le moment d'observer les entorses à cette stratégie implicite d'orientation des règles en examinant les apparitions d'inconsistances triviales. En gros, en expérimenteur, nous avons voulu utiliser REVE tel quel pour simuler cette stratégie, sachant bien que la réécriture est par essence indéterministe. Nous invitons le lecteur sceptique à regarder nos résultats.

Quelles sont les précautions à prendre:

1) La première est de veiller à ce que le système de réécriture ne réduise pas une expression quand la précondition n'est pas valide.

2) On doit rejeter les nouvelles équations dont les deux membres se

première partie

réduisent à "!".

3) On doit trouver les préconditions qui accompagnent les nouvelles règles.

4) Quand REVE s'exécute, les superpositions entre les règles doivent respecter les préconditions.

Les entorses faites par REVE pour les points 1 et 4 vont se déceler très facilement car elles conduisent à des inconsistances triviales.

#### EXEMPLES

1) Proposons les 2 axiomes

LCVID:  $\text{last } o (\text{cons } o [x, <^{\sim}]) = x$

LCONS:  $(\text{DEF } o x) \ \& \ (\text{DEF } o (\text{last } o a)) ::$

$\text{last } o (\text{cons } o [x, a]) = \text{last } o a$

REVE oriente LCONS et LCVID de gauche à droite. On a les deux règles

LCVID:  $\text{last } o (\text{cons } o [x, <^{\sim}]) \Rightarrow x$

LCONS:  $\text{last } o (\text{cons } o [x, a]) \Rightarrow \text{last } o a$

LCONS réduit le membre gauche de LCVID et donne l'équation

$\text{last } o <^{\sim} = x$

qui n'est pas valide.

Que s'est-il passé?

LCONS ne doit pas s'appliquer au membre gauche de LCVID car la précondition

première partie

$\text{DEF } o (\text{last } o <^{\sim})$

n'est pas vérifiée.

On peut remarquer que l'orientation de LCONS ne correspond pas à la stratégie d'utilisation de l'équation LCONS évitant de vérifier la précondition et qui serait celle d'un remplacement de "last o a"

2) Proposons les deux axiomes

APPTRANS:  $\text{DEF } o [\text{trans } o [a, b], \text{trans } o [u, v]] ::$

$\text{append } o [\text{trans } o [a, b], \text{trans } o [u, v]]$

$= \text{trans } o [\text{append } o [a, u], \text{append } o [b, v]]$

APPASS:  $\text{append } o [\text{append } o [a, b], c] = \text{append } o [a, \text{append } o [b, c]]$

Si l'orientation de APPASS est faite dans le sens de droite à gauche, le résultat de la superposition de APPASS à l'occurrence 21, par exemple, donne la nouvelle équation

$\text{append } o [\text{trans } o [a1, b], \text{trans } o [\text{append } o [a2, u], v]]$

$= \text{append } o [\text{trans } o [\text{append } o [a1, a2], b], \text{trans } o [u, v]]$

qui n'est pas valide.

Que s'est-il passé?

La paire critique issue de la superposition de APPASS est

$\text{trans } o [\text{append } o [a1, \text{append } o [a2, u]], \text{append } o [b, v]]$

$= \text{append } o [\text{trans } o [\text{append } o [a1, a2], b], \text{trans } o [u, v]]$

Notons que la validité de cette équation est soumise à la précondition

(1)  $\text{DEF } o [\text{trans } o [\text{append } o [a1, a2], b], \text{trans } o [u, v]]$

issue de APPTRANS.

première partie

REVE a réduit le premier membre en appliquant de nouveau APPTRANS, mais il a tort car la précondition demandée par APPTRANS pour cette réduction

(2)  $\text{DEF } o \text{ [trans } o \text{ [a1,b],trans } o \text{ [append } o \text{ [a2,u],v]]}$

est en contradiction avec (1).

On peut remarquer que l'orientation de APPTRANS est en contradiction avec la stratégie d'utilisation sans vérification de précondition de l'équation APPTRANS.

Malheureusement, il est alors difficile de continuer les essais

On ne peut qu'effectuer des tests incluant séparément les règles en cause et envisager "manuellement" ce qui devrait se passer quand les deux règles sont en présence l'une de l'autre.

Pour expliquer "ce qui se passe", nous avons toujours besoin de connaître la précondition laquelle doit être soumise l'application de la règle.

Si la règle  $g \Rightarrow d$  est issue de l'équation

$C :: g = d \text{ (ou } d = g \text{)},$

nous n'oublions pas la précondition  $C$ .

Nous la notons entre parenthèses après la règle sous la forme

$g \Rightarrow d \text{ SI } (C).$

Quand une précondition doit être vérifiée avant d'utiliser la règle pour réécrire, nous la distinguons en la soulignant.

$g \Rightarrow d \text{ SI } (\underline{\text{DEF } o \text{ p}})$

EXEMPLES On note

première partie

P3:  $?1 \text{ } o \text{ [f,g]} \Rightarrow f \text{ SI } (\underline{\text{DEF } o \text{ g}})$

LCONS:  $\text{last } o \text{ (cons } o \text{ [x,a])} \Rightarrow \text{last } o \text{ a}$

$\text{SI } ((\underline{\text{DEF } o \text{ (last } o \text{ a)}})) \& (\underline{\text{DEF } o \text{ x}}))$

Nous décelons et rejetons les nouvelles équations qui se réduisent à "!"

Elles ne sont donc pas composées d'expressions "correctes" au sens du chapitre II parce que les superpositions donnent une paire de deux expressions équivalentes à "!".

EXEMPLE

La superposition entre

LTL:  $\text{last } o \text{ (distl } o \text{ [x,h])} \Rightarrow [x, \text{last } o \text{ h}]$

et DVID:  $\text{distl } o \text{ [x,<>]} \Rightarrow <> \text{ SI } (\underline{\text{DEF } o \text{ x}})$

donne la paire critique

$\text{last } o \text{ } <> = [x, \text{last } o \text{ } <>]$

dont les deux membres sont extensionnellement égaux à "!".

Cette paire critique doit normalement disparaître.

Comme nous ne donnons pas de règle réduisant cette paire (nous avons décidé au chapitre II de ne pas inclure dans A les axiomes sur "!"), REVE l'ordonne immédiatement en une nouvelle règle.

Notre réaction dans ce cas est alors la suivante. Nous provoquons une interruption de la procédure de Knuth-Bendix et rejetons de telles règles (ou paires critiques). Une nouvelle exécution de la procédure considérera les paires critiques rejetées après le test de l'ensemble du système. Quand

Première partie

les paires critiques rejetées sont prises en compte par la procédure qui commence à les superposer sur les autres règles, c'est que le système a été complété.

REMARQUE Si nous ajoutons les axiomes sur "!", par exemple

D:  $\text{last } o \langle \rangle^- = !^-$

BOT1:  $f \circ !^- = !^-$

BOT2:  $[f, !^-] = !^-$

qui complètent la théorie A avec la classe des expressions A-égales à "!", que se passerait-il?

REVE réduirait "à tort" des axiomes. Par exemple

E3:  $\text{DEF } o \ y :: x^- \circ y = x^-$

se superposerait avec BOT1 et on obtiendrait la nouvelle équation

$x^- = !^-$

qu'il faudrait alors rejeter. J'affirme qu'il est plus facile de rejeter directement toutes les paires critiques qui se réduisent à "!",

Quand une nouvelle équation apparaît, on ne connaît pas sa précondition et il faut la trouver.

Lors de l'exécution de REVE, une nouvelle équation peut provenir soit d'une règle ayant son membre gauche réduit, soit d'une paire critique. Quand une règle à son membre droit réduit, on obtient directement une nouvelle règle. On doit chercher "manuellement" la précondition d'application de la nouvelle équation ou de la nouvelle règle.

première partie

### 2.3. Conclusion: notre méthode est expérimentale

(on peut y trouver l'influence de C. BERNARD)

Parlons un peu de méthodologie avant d'aborder les expériences. Tout d'abord, on ne peut donner tous les axiomes d'un seul coup car il serait impossible de voir ce qui se passe. Nous allons donc partager l'ensemble des axiomes, autrement dit, on essaie d'être modulaire.

Le partage tient compte du découpage des axiomes: axiomes de base, axiomes sur les primitives, sur "ALL", sur "\*",....

A l'intérieur de ce premier découpage, on introduira pratiquement les axiomes un à un pour pouvoir varier l'ordre d'introduction des équations et essayer d'explorer tous les choix possibles d'orientation et d'ordre d'introduction des règles. On fait, en quelque sorte, du "backtracking" dans l'addition des axiomes.

Dans le paragraphe suivant, nous allons commenter notre expérimentation en motivant nos choix. La lecture de ce paragraphe pourra paraître fastidieuse, mais la lecture exhaustive n'est pas nécessaire. C'est un document sur l'utilisation du système REVE (c'est à dire d'une implémentation particulière de la procédure de Knuth-Bendix) pour obtenir des systèmes de réécriture convergents à partir d'un ensemble d'axiomes important comme nos axiomes de la FP-algèbre. C'est une expérience originale car c'est un des cas ([Che83], [Cho85], [Cos85]) où la procédure de Knuth-Bendix a été appliquée sur un grand nombre d'axiomes.

Une lecture complète n'est pas indispensable pour comprendre la suite.

première partie

Il suffit de lire, pour chaque classe d'axiomes, les conclusions de l'expérience soit

a) au paragraphe 3.1: Les axiomes de base sans les axiomes sur la conditionnelle.

b) au paragraphe 3.2: Les axiomes sur les primitives sur les séquences. Nous avons alors distingué les axiomes sur les diverses primitives: "cons", "distl", "trans", "iota" et enfin "append".

c) au paragraphe 3.3: Les axiomes sur "ALL".

d) au paragraphe 3.4: Les axiomes sur "x".

e) au paragraphe 3.5: Les axiomes sur "WH". L'expérience suit à peu de chose près celle des axiomes sur "x".

f) et enfin au paragraphe 3.6: Les axiomes sur la conditionnelle qui peuvent s'ajouter à chacune des expériences précédentes.

De plus, nous avons récapitulé les principaux systèmes de réécriture obtenus dans l'annexe II.

### 3. L'expérimentation

Très logiquement, nous commencerons par le noyau des axiomes de base qui plus tard se superposeront avec tous les autres axiomes de la FP- algèbre.

#### 3.1. Les axiomes de base

première partie

Chacun des deux axiomes E1 et E2 peut être orienté dans les deux sens ce qui donne quatre cas à tester.

#### a) Les axiomes E1 et E2

Nous présentons à REVE les deux équations

$$E1: (f \circ g) \circ h = f \circ (g \circ h)$$

$$E2: [f, g] \circ h = [f \circ h, g \circ h]$$

à orienter de gauche à droite. Pour orienter E1, il faut choisir un statut LR pour "o". Pour orienter E2 il faut choisir la précedence  $o > [ ]$ . On obtient alors un système convergent formé des deux règles

```

+++++
+ S-E1E2 o(LR)>[] +
+ ***** +
+ +
+ E1: (f o g) o h => f o (g o h) +
+ +
+ E2: [f,g] o h => [f o h,g o h] +
+ +
+++++

```

Nous changeons le sens de E2 en choisissant la précedence  $[ ] > o$ . Les superpositions de E1 sur E2 aux occurrences 1 et 2 donnent alors deux paires critiques

$$[f1 \circ (f2 \circ h), g \circ h] = [f1 \circ f2, g] \circ h$$

$$[f \circ h, g1 \circ (g2 \circ h)] = [f, g1 \circ g2] \circ h$$

On considère la première paire qui ne se réduit pas et fournit une nouvelle règle dans le sens gauche droite.

Moyennant quoi, le processus diverge ultérieurement car, sur cette nouvelle règle, les superpositions de E1 aux occurrences 22 et 2 donnent encore deux paires critiques dont l'une est

Première partie

$$[f_1 \circ (f_2 \circ (f_3 \circ h)), g \circ h] = [f_1 \circ (f_2 \circ f_3), g] \circ h$$

A partir de cette nouvelle équation, le même phénomène se poursuit indéfiniment.

A part E1, aucune conséquence équationnelle de A ne peut détruire ces paires critiques: On abandonne ce choix.

Nous changeons maintenant le sens de E1 en donnant à "o" le statut RL et gardons le sens initial de E2 avec la précedence  $o > [ ]$ .

Une paire critique est obtenue par superposition de E2 sur E1 à l'occurrence 2

$$k \circ [f \circ h, g \circ h] = (k \circ [f, g]) \circ h$$

Elle s'oriente de gauche à droite. Deux paires critiques sont obtenues par superposition de E2 sur cette nouvelle règle aux occurrences 21 et 22:

$$k \circ [[f_1 \circ h, g_1 \circ h], g_2 \circ h] = (k \circ [[f_1, g_1], g_2]) \circ h$$

$$k \circ [f_1 \circ h, [f_2 \circ h, g_2 \circ h]] = (k \circ [f_1, [f_2, g_2]]) \circ h$$

Considérons déjà la première équation qui s'oriente de gauche à droite:

Trois paires critiques sont encore obtenues par superposition de E2 sur cette nouvelle règle aux occurrences 211, 212, 22. Citons la première

$$k \circ [[[f_1 \circ h, g_1 \circ h], g_2 \circ h], g_3 \circ h] = (k \circ [[[f_1, g_1], g_2], g_3]) \circ h$$

Le processus continue indéfiniment.

A part E2, aucune conséquence équationnelle ne peut détruire ces paires critiques: Ce choix est abandonné.

première partie

La dernière possibilité est de renverser de nouveau le sens de E2 en choisissant la précedence  $[ ] > o$ . On obtient alors un système convergent avec les deux règles

```

+++++
+ S-REV-E1E2 [] > o(RL) +
+ ++++++ +
+ +
+ E1: f o (g o h) => (f o g) o h +
+ +
+ E2: [f o h, g o h] => [f, g] o h +
+ +
+++++

```

#### b) Autres axiomes de base

Avec ces axiomes, il n'y a pas de problème de choix d'orientation. Ils doivent tous être orientés dans le sens gauche droite. En les ajoutant aux deux règles de S-E1E2, on obtient un système convergent S-BASE

```

+++++
+ S-BASE o(LR) > [] +
+ ++++++ +
+ +
+ S-E1E2 et +
+ +
+ +
+ E3: x- o y => x- SI (DEF o y) +
+ +
+ P1: id o f => f +
+ +
+ P2: f o id => f +
+ +
+ P31: ?1 o [f, g] => f SI (DEF o [f, g]) +
+ +
+ P32: ?2 o [f, g] => g SI (DEF o [f, g]) +
+ +
+++++

```

Reprenons maintenant les deux règles de S-REV-E1E2, et ajoutons les deux équations

première partie

$$P3_1: ?1 \circ [f, g] = f \text{ SI } (DEF \circ [f, g])$$

$$P3_2: ?2 \circ [f, g] = g \text{ SI } (DEF \circ [f, g])$$

qui s'orientent de gauche à droite.

La règle  $P3_1$  se superpose sur la règle E1 à l'occurrence 2 et sur la règle E2 aux occurrences 1 et 2 donnant les 3 paires critiques

$$(h \circ f) = (h \circ ?1) \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

$$[f, k \circ [f, g]] = [?1, k] \circ [f, g]$$

$$[k \circ [f, g], f] = [k, ?1] \circ [f, g]$$

La première s'oriente de droite à gauche et donne une nouvelle règle r1.

Il y a 3 paires critiques analogues avec la règle  $P3_2$ . La première

$$(h \circ g) = (h \circ ?2) \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

s'oriente de droite à gauche en une règle r2.

Les superpositions de r1 et r2 avec la règle E2 donnent 4 paires critiques, celles dues à r1 sont

$$[h \circ f, k \circ [f, g]] = [h \circ ?1, k] \circ [f, g]$$

$$[k \circ [f, g], h \circ f] = [k, h \circ ?1] \circ [f, g]$$

Ces équations s'orientent de gauche à droite en étendant la précedence avec  $[\_]>?1$  (ou  $o>?1$ ) qui sont les deux choix proposés par REVE.

Les superpositions de ces nouvelles règles donnent 2 paires critiques avec les deux règles P3 et avec r1 et r2.

$$[f, f] = [?1, ?1] \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

$$[f, g] = [?1, ?2] \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

$$[f, h \circ f] = [?1, h \circ ?1] \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

$$[f, h \circ g] = [?1, h \circ ?2] \circ [f, g] \text{ SI } (DEF \circ [f, g])$$

première partie

qui vont s'orienter en de nouvelles règles de droite à gauche, et tout recommencera comme avec les deux règles P3 pour ces nouvelles règles.

On ne trouve pas de conséquences équationnelles autres que P3 et E2 pour réduire ces paires critiques. Ce choix est abandonné.

S-BASE servira de noyau à tous nos systèmes de réécriture.

### 3.2. Les axiomes sur les primitives

L'axiome sur "last" et "t1"

Il s'oriente de gauche à droite donnant la règle

$$LTL: \text{last} \circ t1 \Rightarrow \text{last} \text{ SI } (DEF \circ (\text{last} \circ t1))$$

et on obtient une nouvelle règle

$$LTL': \text{last} \circ (t1 \circ h) \Rightarrow \text{last} \circ h \text{ SI } (DEF \circ (\text{last} \circ (t1 \circ h)))$$

par superposition sur E1.

On obtient un système convergent S-LTL.

$$S-LTL \quad o(LR)>[\_]$$

\*\*\*\*\*

S-BASE et

\*\*\*\*\*

$$LTL: \text{last} \circ t1 \Rightarrow \text{last} \text{ SI } (DEF \circ (\text{last} \circ t1))$$

$$LTL': \text{last} \circ (t1 \circ h) \Rightarrow \text{last} \circ h \text{ SI } (DEF \circ (\text{last} \circ (t1 \circ h)))$$

\*\*\*\*\*

Les axiomes sur "cons"

première partie

● On introduit les axiomes

HCONS: DEF o (cons o [x,a]) :: hd o (cons o [x,a]) = x

LCONS: (DEF o x) & (DEF o (last o a)) ::

last o (cons o [x,a]) = last o a

TLCONS: DEF o (cons o [x,a]) :: tl o (cons o [x,a]) = a

qui s'orientent de gauche à droite. HCONS et LCONS ne posent pas de problème pour leur orientation puisque la précondition est sous-terme du membre gauche.

Mais ce n'est pas le cas de LCONS

LCONS: last o (cons o [x,a]) => last o a SI (DEF o x) & (DEF o (last o a))

et il faudra observer que les réécritures faites par la règle LCONS ne sont faites que lorsque l'on a "DEF o (last o a)". LCONS ne se superpose sur aucune règle et n'en réduit aucune.

On obtient un système convergent S-CONSO.

S-CONSO o(LR)>[\_\_]  
\*\*\*\*\*  
S-LTL et  
\*\*\*\*  
HCONS: hd o (cons o [x,a]) => x SI (DEF o (cons o [x,a]))  
TLCONS: tl o (cons o [x,a]) => a SI (DEF o (cons o [x,a]))  
LCONS: last o (cons o [x,a]) => last o a SI ((DEF o (last o a)) & (DEF o x))  
\*\*\*\*\*

● On introduit CONS

première partie

CONS: DEF o (tl o a) :: cons o [hd o a, tl o a] = a

P2 se superpose sur CONS, ce qui produit une paire critique

cons o [hd,tl] = id SI (DEF o tl)

Pour l'orienter, la précedence est étendue avec o>id et on obtient la nouvelle règle

CONS': cons o [hd,tl] => id SI (DEF o tl)

On obtient un système convergent S-CONS.

S-CONS o(LR)>{[\_\_],id}  
\*\*\*\*\*  
S-CONSO et  
\*\*\*\*\*  
CONS: cons o [hd o a,tl o a] => a SI (DEF o (tl o a))  
CONS': cons o [hd,tl] => id SI (DEF o tl)  
\*\*\*\*\*

● Il reste l'axiome LCVID

LCVID: last o (cons o [x,<>~]) = x

REVE réduit LCVID en utilisant LCONS et produit l'équation

<>~ = x

Nous observons cette inconsistance. La réduction par LCONS ne doit pas être faite si on tient compte de la précondition de LCONS. Mais si nous enlevons l'équation LCONS. REVE se termine.

première partie

On peut affirmer que, si on ajoute la règle LCVID aux systèmes conditionnels S-CONS et S-CONSO, on obtient des systèmes de réécriture convergents S-CVIDO et S-CVID.

$o(LR) > [\_]$  S-CVIDO ou S-CVID (avec  $o > id$ )  
\*\*\*\*\*

S-CONSO ou S-CONS et

\*\*\*\*\*

LCVID:  $last\ o\ (cons\ o\ [x, <\_]) \Rightarrow x$

\*\*\*\*\*

#### Les axiomes sur "trans"

Rappelons que la primitive "trans" appliquée à deux séquences de même longueur donne la séquence des couples de leurs termes.

● On introduit les axiomes

HTRANS:  $DEF\ o\ (trans\ o\ [a,b])::$

$hd\ o\ (trans\ o\ [a,b]) = [hd\ o\ a, hd\ o\ b]$

LTRANS:  $DEF\ o\ (trans\ o\ [a,b])::$

$last\ o\ (trans\ o\ [a,b]) = [last\ o\ a, last\ o\ b]$

TLTRANS:  $tl\ o\ (trans\ o\ [a,b]) = trans\ o\ [tl\ o\ a, tl\ o\ b]$

TCONS:  $cons\ o\ [[x,y], trans\ o\ [a,b]]$

$= trans\ o\ [cons\ o\ [x,a], cons\ o\ [y,b]]$

Les axiomes HTRANS, LTRANS s'orientent de gauche à droite.

L'orientation de TLTRANS, de gauche à droite réclame une extension de la précedence avec  $tl > trans$ .

REVE ne tient pas compte immédiatement des axiomes TCONS et TLCONS et commence à calculer les paires critiques avec la règle CONS. On obtient

première partie

alors des nouvelles règles. Une des paires critiques

$cons\ o\ [[hd,hd], trans\ o\ [tl,tl]] = trans\ o\ [id,id]\ SI\ (DEF\ o\ tl)$

demande une augmentation de la précedence.

On choisit  $cons > trans$  et elle s'oriente de gauche à droite.

L'arrivée de TCONS, qui s'oriente de gauche à droite, permet de détruire toutes les nouvelles règles. On obtient un système convergent S-TRANS.

On peut aussi choisir  $trans > cons$  et TCONS est orienté dans l'autre sens. REVE diverge avec les superpositions de la règle CONS sur TCONS. Par contre, avec S-CVIDO qui ne contient pas CONS, on obtient un système convergent S-TRANSO.

Première partie

```

 a(LR)>[_],tl>trans
S-TRANSO (avec trans>cons) ET S-TRANS (avec o>id,cons>trans)

S-CVIDO ou S-CVID et

HTRANS: hd o (trans o [a,b]) => [hd o a,hd o b] SI (DEF o (trans o [a,b]))
LTRANS: last o (trans o [a,b]) => [last o a,last o b]
 SI (DEF o (trans o [a,b]))
TLTRANS: tl o (trans o [a,b]) => trans o [tl o a,tl o b]

pour S-TRANSO (avec S-CVIDO)

TCONS: trans o [cons o [x,a],cons o [y,b]] => cons o [[x,y],trans o [a,b]]
pour S-TRANS (avec S-CVID)

TCONS: cons o [[x,y],trans o [a,b]] => trans o [cons o [x,a],cons o [y,b]]

```

- On ajoute TVID

TVID: trans o [ $\langle \rangle^-, \langle \rangle^-$ ] =  $\langle \rangle^-$

qui s'oriente de gauche à droite.

TVID crée par superposition avec HTRANS, TLTRANS et LTRANS des paires critiques équivalentes à "!" que l'on rejette.

Dans le cas de S-TRANSO, REVE se termine. On obtient un système convergent S-TVIDO.

Dans le cas de S-TRANS, une paire critique est obtenue avec TCONS

première partie

TVID': cons o [[x,y], $\langle \rangle^-$ ]  
 == trans o [cons o [x, $\langle \rangle^-$ ],cons o [y, $\langle \rangle^-$ ]]

Elle s'oriente de gauche à droite ne créant pas d'autres paires. Le système obtenu S-TVID est convergent.

S-TVIDO et S-TVID

```

S-TRANSO ou S-TRANS et

TVID: trans o [$\langle \rangle^-, \langle \rangle^-$] => $\langle \rangle^-$
Pour S-TVID (avec S-TRANS)

TVID': cons o [[x,y], $\langle \rangle^-$] => trans o [cons o [x, $\langle \rangle^-$],cons o [y, $\langle \rangle^-$]]

```

Les axiomes sur "distl"

Rappelons que la primitive "distl" permet de coupler un même objet à tous les termes d'une même séquence.

- On introduit les axiomes

HDISTL: hd o (distl o [x,a]) = [x,hd o a]

LDISTL: last o (distl o [x,a]) = [x,last o a]

TLDISTL: tl o (distl o [x,a]) = distl o [x,tl o a]

DCONS: cons o [[x,y],distl o [x,a]] = distl o [x,cons o [y,a]]

Les axiomes HDISTL, LDISTL sont orientés de gauche à droite. L'orientation de TLDISTL de gauche à droite réclame une extension de la précédence avec tl>distl.

REVE calcule les paires critiques sans tenir compte de DCONS. Une

première partie

nouvelle équation

$$\text{cons } o \{ [x, \text{hd}], \text{distl } o [x, \text{tl}] \} = \text{distl } o [x, \text{id}] \text{ SI } (\text{DEF } o \text{ tl})$$

réclame une extension de la précedence.

Si on choisit  $\text{cons} > \text{distl}$ , DCONS sera orientée de gauche à droite et les nouvelles règles sont détruites. Complétant les règles de S-TVID, on obtient un système convergent S-DISTL.

On peut choisir d'orienter DCONS dans l'autre sens grâce à la précedence  $\text{distl} > \text{cons}$ . Dans le cas du système S-DISTLO (ne contenant pas CONS), on obtient encore un système convergent S-DISTLO.

$o(\text{LR}) > [\_, \text{tl}] > \{ \text{trans}, \text{distl} \}$   
 S-DISTLO  $\{ \text{trans}, \text{distl} \} > \text{cons}$  ET S-DISTL  $o > \text{id}, \text{cons} > \{ \text{trans}, \text{distl} \}$   
 \*\*\*\*\*

S-TVID ou S-TVID et

\*\*\*\*\*

HDISTL:  $\text{hd } o (\text{distl } o [x, a]) \Rightarrow [x, \text{hd } o a]$

LDISTL:  $\text{last } o (\text{distl } o [x, a]) \Rightarrow [x, \text{last } o a]$

TLDISTL:  $\text{tl } o (\text{distl } o [x, a]) \Rightarrow \text{distl } o [x, \text{tl } o a]$

\*\*\*\*\*

Pour S-DISTLO (avec S-TVID)

DCONS:  $\text{distl } o [x, \text{cons } o [y, a]] \Rightarrow \text{cons } o [[x, y], \text{distl } o [x, a]]$

Pour S-DISTL (avec S-TVID)

DCONS:  $\text{cons } o [[x, y], \text{distl } o [x, a]] \Rightarrow \text{distl } o [x, \text{cons } o [y, a]]$

\*\*\*\*\*

● On ajoute alors DVID

première partie

DVID:  $\text{DEF } o x :: \text{distl } o [x, <>] = <>$

Avec HDISTL, LDISTL et TLDISTL, on obtient des paires critiques équivalentes à "!" que l'on rejette.

Dans le cas de S-DISTL, par superposition sur DCONS, on obtient la nouvelle équation

TDVID:  $\text{trans } o [\text{cons } o [x, <>], \text{cons } o [y, <>]]$

$= \text{distl } o [x, \text{cons } o [y, <>]]$

qui s'oriente de gauche à droite si on étend la précedence avec  $\text{trans} > \text{distl}$ . Le membre gauche de TVID' est réduit et TVID' disparaît. Mais ensuite REVE diverge par des superpositions sur TCONS et DCONS. Le même phénomène se produit si on oriente dans l'autre sens ( $\text{distl} > \text{trans}$ ).

Cependant, DVID s'ajoute au système S-DISTLO et on obtient un système convergent S-TD.

première partie

```

o(LR)>[], tl>{trans,distl}, {trans,distl}>cons
S-TD

S-BASE ou S-LTL

HCONS: hd o (cons o [x,a]) => x SI (DEF o (cons o [x,a]))
LCONS: last o (cons o [x,a]) => last o a SI((DEF o (last o a)) & (DEF o x))

TLCONS: tl o (cons o [x,a]) => tl o a SI (DEF o (cons o [x,a]))
HTRANS: hd o (trans o [a,b]) => [hd o a,hd o b]
 SI (DEF o (trans o [a,b]))
LTRANS: last o (trans o [a,b]) => [last o a,last o b]
 SI (DEF o (trans o [a,b]))
TLTRANS: tl o (trans o [a,b]) => trans o [tl o a,tl o b]
TCONS: trans o [cons o [x,a],cons o [x,b]] => cons o [[x,y],trans o [a,b]]
HDISTL: hd o (distl o [x,a]) => [x,hd o a]
LDISTL: last o (distl o [x,a]) => [x,last o a]
TLDISTL: tl o (distl o [x,a]) => distl o [x,tl o a]
DCONS: distl o [x,cons o [y,a]] => cons o [[x,y],distl o [x,a]]
CVID: last o (cons o [x,<>]) = x
TVID: trans o [<>,<>] => <>
DVID: distl o [x,<>] => <> SI (DEF o x)

```

\*\*\*\*\*

Nous continuons avec le système S-TD et nous abandonnons l'axiome CONS  
 ~~~~~

#### Les axiomes sur "iota"

On introduit

première partie

```

HIOTA: DEF o (iota o n) :: hd o (iota o n) = 1-
LIOTA: DEF o (iota o n) :: last o (iota o n) = n

```

qui s'orientent de gauche à droite. L'orientation de LIOTA exige une extension de la précédence avec o>id. Celle de HIOTA demande aussi d'étendre la précédence avec o>l et o><sup>-</sup>.

Deux nouvelles règles

```

HIOTA': hd o (iota o n) => 1- SI (DEF o (iota o n))
LIOTA': last o (iota o n) => SI (DEF o (iota o n))

```

sont issues de superpositions sur E1. Le système obtenu S-IOTA est alors convergent.

```

o(LR)>[],id,l,-,tl>{trans,distl}
S-IOTA {trans,distl}>cons

S-TD et

HIOTA: hd o iota => 1- SI (DEF o iota)
HIOTA': hd o (iota o n) => 1- SI (DEF o (iota o n))
LIOTA: last o iota => id SI (DEF o iota)
LIOTA': last o (iota o n) => n SI (DEF o (iota o n))

```

#### Les axiomes sur "append"

- On introduit les axiomes

Première partie



L'équation APPASS réécrite avec "A" est ordonnée de gauche à droite avec le statut LR pour "A" et on obtient un système convergent S-APP'.

Le morphisme des FP-termes dans les termes  $T(S-\{\text{append}\}+A, V)$  met les réécritures en bijection et ainsi une réécriture avec S-APPO correspond à une réécriture avec S-APP' et réciproquement (voir la deuxième partie). S-APPO est donc convergent.

● On introduit

APPDISTL:  $\text{append } o [\text{distl } o [x, a], \text{distl } o [x, b]]$

$= \text{distl } o [x, \text{append } o [a, b]]$

qui s'oriente droite à gauche (on a  $\text{distl} > \text{append}$ ). La règle APPDISTL' est alors détruite par APPDISTL.

On obtient un système convergent S-APP.  
 ~~~~~

```

o(LR)>{[_],id,1,-},tl>{trans,distl,append},cons>append
 {trans,distl}>append
S-APP {trans,distl}>cons
~~~~~
S-IOTA      et
~~~~~
LAPP: last o (append o [a,b]) => last o b
 SI ((DEF o (last o b)) & (DEF o (append o [a,b])))
~~~~~
HAPP: hd o (append o [a,b]) => hd o a
      SI ((DEF o (hd o a)) & (DEF o (append o [a,b])))
~~~~~
TLAPP: tl o (append o [a,b]) => append o [tl o a, b]
 SI (DEF o (tl o a))
~~~~~
APPCONS: cons o [x, append o [a,b]] => append o [cons o [x,a], b]
APPASS:  append o [append o [a,b], c] => append o [a, append o [b,c]]
APPDISTL: distl o [x, append o [a,b]] => append o [distl o [x,a], distl o [x,b]]
APPTRANS': trans o [append o [cons o [x,a], b], cons o [y,c]]
          => cons o [[x,y], trans o [append o [a,b], c]]
APPTRANS'S: trans o [cons o [x,a], append o [cons o [y,b], c]]
          => cons o [[x,y], trans o [a, append o [b,c]]]
APPTRANS'': trans o [append o [cons o [x,a], b], append o [cons o [y,u], v]]
          => cons o [[x,y], [trans o [append o [a,b], append o [u,v]]]
~~~~~

```

● On introduit l'axiome APPTRANS

APPTRANS:  $\text{DEF } o [\text{trans } o [a,b], \text{trans } o [u,v]] ::$   
 $\text{append } o [\text{trans } o [a,b], \text{trans } o [u,v]]$   
 $= \text{trans } o [\text{append } o [a,u], \text{append } o [b,v]]$

APPTRANS s'oriente de droite à gauche (on a  $\text{trans} \triangleright \text{append}$ ). La règle APPTRANS" est détruite par APPTRANS.

On doit rejeter des nouvelles équations non valides provenant des superpositions de APPASS sur APPTRANS (en exemple au paragraphe 2.2). Il nous faut cependant observer ce qui s'est passé. La superposition de APPASS sur APPTRANS à l'occurrence 21 donne la paire critique

$$\begin{aligned} (P) \quad & \text{trans o [append o [a1, append o [a2, u]], append o [b, v]]} \\ &= \text{append o [trans o [append o [a1, a2], b], trans o [u, v]]} \\ \text{SI } & (\text{DEF o [trans o [append o [a1, a2], b], trans o [u, v]]} \end{aligned}$$

Mais REVE réduit le membre droit en utilisant APPTRANS alors que la précondition

"DEF o [trans o [a1, b], trans o [append o [a2, u], v]]" n'est pas vérifiée.

Il donne alors une nouvelle équation

$$\begin{aligned} & \text{append o [trans o [a1, b], trans o [append o [a2, u], v]]} \\ &= \text{append o [trans o [append o [a1, a2], b], trans o [u, v]]} \end{aligned}$$

qui n'est pas valide.

Si l'on rejette ces équations, REVE se termine. Mais le système de réécriture obtenu n'est convergent que si l'on tient compte des nouvelles équations (P). On ne peut pas faire le test avec REVE. Mais on peut voir que les superpositions de APPASS vont engendrer une infinité de règles nouvelles à partir de (P). Nous les étudierons dans la deuxième partie.

- On introduit les axiomes

APPVID: DEF o (append o [a, <>]) :: append o [a, <>] = a

APPVIDS: DEF o (append o [<>, b]) :: append o [<>, b] = b

première partie

qui s'orientent de gauche à droite.

Par superposition avec APPCONS, on obtient la paire critique

$$\text{cons o [x, b] = append o [cons o [x, <>], b]}$$

qui n'est pas orientable avec REVE.

Si on oriente "manuellement" de droite à gauche, REVE diverge avec les superpositions sur la règle APPCONS.

L'orientation de gauche à droite engendre une chaîne infinie de réécriture.

$$\begin{aligned} & \text{cons o [x, b] } \Rightarrow \text{append o [cons o [x, <>], b]} \\ \Rightarrow & \text{append o [append o [(cons o [x, <>], <>), b] } \Rightarrow \dots \end{aligned}$$

- On essaie d'orienter APPCONS différemment (append > cons)

$$\text{APPCONS: append o [cons o [x, a], b] } \Rightarrow \text{cons o [x, append o [a, b]]}$$

On obtient un système convergent avec S-IOTA et les axiomes HAPP, LAPP, TLAPP, APPCONS, APPDISTL (avec  $\text{distl} \triangleright \text{append}$ ) et APPASS.

APPVID et APPVIDS s'introduisent dans ce système et REVE se termine quand on rejette les paires critiques réductibles à "!" dues aux superpositions avec HAPP, LAPP et TLAPP.

On obtient un système convergent S-PRIM. Mais si on ajoute APPTRANS, REVE diverge à cause des superpositions de APPCONS.

première partie

```

o>{[_],id,l,-},tl>{trans,distl,append}
{append,distl,trans}>cons, distl>append
S-PRIM

S-IOTA et

HAPP: hd o (append o [a,b]) => hd o a
 SI ((DEF o (hd o a)) & (DEF o (append o [a,b])))
LAPP: last o (append o [a,b]) => last o b
 SI ((DEF o (last o b)) & (DEF o (append o [a,b])))
TLAPP: tl o (append o [a,b]) => append o [tl o a,b] SI (DEF o (tl o a))
APPCONS: append o [cons o [x,a],b] => cons o [x,append o [a,b]]
APPASS: append o [append o [a,b],c] => append o [a,append o [b,c]]
APPDISTL: distl o [x,append o [a,b]] =>
 append o [distl o [x,a],distl o [x,b]]
APPVID: append o [a,<~] => a SI (DEF o (append o [a,<~]))
APPVIDS: append o [<~],a] => a SI (DEF o (append o [<~],a]))

```

④ On revient en arrière et on renverse l'orientation de APPCONS.

On introduit un opérateur pour diviser la paire critique non orientable  
 ~~~~~

cons o [x,b] = append o [cons o [x,<~],b].

Soit un opérateur "L" tel que

DEFLIST: L(x) = cons o [x,<~].  
 ~~~~~

première partie

Cet axiome DEFLIST s'oriente de droite à gauche avec la précedence o>L.

On obtient une nouvelle règle

LIST: L(x) o y => L(x o y)

par superposition sur E1.

Puis les règles

HLIST: hd o L(x) => x

TLLIST: tl o L(x) => <~ SI (DEF o x)

par superposition sur HCONS et TLCONS.

La paire critique obtenue par superposition sur LCONS est à rejeter.

Mais LCVID devient

LCVID: last o L(x) => x

Une superposition sur la règle TCONS donne la nouvelle règle

TLIST: trans o [L(x),L(y)] => L([x,y])

Les paires critiques provenant des autres superpositions sont à rejeter.

On a une nouvelle règle par superposition sur la règle DCONS

DL: distl o [x,L(y)] => L([x,y])

Les règles sur IOTA n'apportent pas de nouvelles règles. On a une nouvelle règle DEFCONS par superposition sur APPCONS

DEFCONS: cons o [x,a] => append o [L(x),a]

première partie

DEFCONS détruit HCONS, LCONS. DEFCONS détruit aussi TLCONS en utilisant APPVID. DEFCONS détruit APPCONS. DEFCONS détruit DCONS en utilisant APPDISTL.

DEFCONS réécrit TCONS

TCONS: trans o [append o [L(x)),a],append o [L(y),b]]  
 => append o [[L(x),L(y)],trans o [a,b]]

DEFCONS détruit alors APPTRANS', APPTRANS'S et APPTRANS'' en utilisant TCONS.

APPVID et APPVIDS s'introduisent alors sans difficulté. DEFLIST disparaît.

APPTRANS détruit TCONS et si l'on rejette les paires critiques non valides provenant des superpositions de APPASS, REVE se termine.

Nous appelons S-PRIMITIVE ce système. Comme on néglige les paires critiques telles que (P) qui sont issues des superpositions de APPASS sur APPTRANS, le système S-PRIMITIVE n'est pas convergent. Cependant nous trouverons le moyen, au cours de la deuxième partie, de les introduire en substituant à APPTRANS une méta-règle. Aussi nous choisissons de poursuivre les essais avec S-PRIMITIVE.

#### REMARQUES

- 1) Sans APPTRANS, on obtient un système convergent.
- 2) S-PRIMITIVE inclut tous les axiomes sur les primitives sur les séquences sauf

CONS: DEF o (cons o [hd o a,tl o a]) :: cons o [hd o a,tl o a] = a

première partie

qui se réduit en

CONS: append o [L(hd o a),tl o a] => a

SI (DEF o (append o [L(hd o a),tl o a]))

et fait diverger REVE.

première partie

```

+++++
+ o(LR)>{[_],id,-,1,L}
+ {trans,dist1}> cons > append
+ t1>{trans,cons,dist1,append}
+ S-PRIMITIVE
+ *****
+ S-BASE et
+ *****
+ LTL: last o t1 => last SI (DEF o (last o t1))
+ LTL': last o (t1 o h) => last o h SI (DEF o (last o (t1 o h)))
+ LIST: L(x) o y => L(x o y)
+ HLIST: hd o L(x) => x
+ LCVID: last o L(x) => x
+ TLLIST: t1 o L(x) => <>~ SI (DEF o x)
+ HTRANS: hd o (trans o [a,b]) => [hd o a,hd o b]
+ SI (DEF o (trans o [a,b]))
+ LTRANS: last o (trans o [a,b]) => [last o a,last o b]
+ SI (DEF o (trans o [a,b]))
+ TLTRANS: t1 o (trans o [a,b]) => trans o [t1 o a,t1 o b]
+ TLIST: trans o [L(x),L(y)] => L([x,y])
+ HDISTL: hd o (dist1 o [x,a]) => [x,hd o a]
+ LDISTL: last o (dist1 o [x,a]) => [x,last o a]
+ TLDISTL: t1 o (dist1 o [x,a]) => dist1 o [x,t1 o a]
+ DLIST: dist1 o [x,L(y)] => L([x,y])
+ TVID: trans o [<>~,<>~] => <>~
+ DVID: dist1 o [x,<>~] => <>~ SI (DEF o x)
+ HIOTA: hd o iota => 1~ SI (DEF o iota)
+ HIOTA': hd o (iota o n) => 1~ SI (DEF o (iota o n))
+ LIOTA: last o iota => id SI (DEF o iota)
+

```

première partie

```

+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))
+ HAPP: hd o (append o [a,b]) => hd o a
+ SI ((DEF o (hd o a)) & (DEF o (append o [a,b])))
+ LAPP: last o (append o [a,b]) => last o b
+ SI ((DEF o (last o b)) & (DEF o (append o [a,b])))
+ TLAPP: t1 o (append o [a,b]) => append o [t1 o a,b] SI (DEF o (t1 o a))
+ DEFCONS: cons o [x,a] => append o [L(x),a]
+ APPASS: append o [append o [a,b],c] => append o [a,append o [b,c]]
+ APPDISTL: dist1 o [x,append o [a,b]]
+ => append o [dist1 o [x,a],dist1 o [x,b]]
+ APPTANS: trans o [append o [a,b],append o [c,d]]
+ => append o [trans o [a,c],trans o [b,d]]
+ SI (DEF o [trans o [a,c],trans o [b,d]])
+ APPVID: append o [a,<>~] => a SI (DEF o (append o [a,<>~]))
+ APPVIDS: append o [<>~,a] => a SI (DEF o (append o [<>~,a]))
+
+++++

```

### 3.3. Les axiomes sur "ALL"

#### Essai avec ALLALL

Ajoutons au système S-E1E2 l'équation

ALLALL: ALL f o ALL g = ALL(f o g)

⊗ Orientation de ALLALL de gauche à droite

Il faut étendre la précedence avec o>ALL.

première partie

Une paire critique est obtenue par superposition de ALLALL sur E1 à l'occurrence 1

$$\text{ALLALL}' : \text{ALL}(f \circ g) \circ h = \text{ALL } f \circ (\text{ALL } g \circ h)$$

qui s'oriente de gauche à droite et donne des paires critiques avec E2, ALLALL et elle-même:

$$\text{ALL}[f \circ h, g \circ h] \circ k = \text{ALL}[f, g] \circ (\text{ALL } h \circ k)$$

$$\text{ALL}(\text{ALL}(f \circ g)) \circ k = \text{ALL } f \circ (\text{ALL } g \circ k)$$

$$\text{ALL}(\text{ALL } f) \circ (\text{ALL}(\text{ALL}(g) \circ (\text{ALL}(k) \circ h)) = \text{ALL}(\text{ALL}(f \circ g)) \circ (\text{ALL } k \circ h)$$

La dernière paire critique est orientée de droite à gauche.

La nouvelle règle provoque de nouveau des paires critiques analogues avec E2, ALLALL, et les deux nouvelles règles (dont elle-même) et le même processus va recommencer indéfiniment.

#### ● Orientation de ALLALL en sens inverse

La précédence doit être étendue avec ALL>o.

On a une paire critique par superposition de E2 sur ALLALL à l'occurrence 1

$$\text{ALL}[f \circ h, g \circ h] = \text{ALL}[f, g] \circ \text{ALL } h$$

qui s'oriente de gauche à droite. E1 et E2 se superposent sur cette règle aux occurrences 11 et 12. On obtient des paires critiques

$$\text{ALL}[f1 \circ (f2 \circ h), g \circ h] = \text{ALL}[f1 \circ f2, g] \circ \text{ALL } h$$

$$\text{ALL}[f, g1 \circ (g2 \circ h)] = \text{ALL}[f, g1 \circ g2] \circ \text{ALL } h$$

$$\text{ALL}[[f \circ h, g \circ h], k \circ h] = \text{ALL}[[f, g], k] \circ \text{ALL } h$$

$$\text{ALL}[k \circ h, [f \circ h, g \circ h]] = \text{ALL}[k, [f, g]] \circ \text{ALL } h$$

première partie

qui s'orientent de gauche à droite. E1 et E2 vont de nouveau se superposer sur ces règles et le processus continue indéfiniment.

#### ● Nous revenons en arrière

$$\text{ALLALL} : \text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$$

engendre la paire critique

$$\text{ALLALL}' : \text{ALL } f \circ (\text{ALL } g \circ h) = \text{ALL}(f \circ g) \circ h$$

L'ordre implémenté dans REVE oriente cette paire de droite à gauche ce qui provoque la divergence.

Pourtant la réécriture par des règles

$$\text{ALLALL} : \text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$$

$$\text{ALLALL}' : \text{ALL } f \circ (\text{ALL } g \circ h) \Rightarrow \text{ALL}(f \circ g) \circ h$$

est seule susceptible de réduire le nombre de séquences intermédiaires.

Pour pouvoir orienter ALLALL' dans ce sens, nous orientons les règles "manuellement".

Nous orientons manuellement E1, E2 et ALLALL de gauche à droite. La paire critique ALLALL' est alors orientée manuellement dans le sens voulu.

Avec chacun des systèmes S-BASE et S-PRIMITIVE (orientés manuellement), REVE se termine.

#### Introduction des axiomes

$$\text{HALL} : \text{DEF } o \text{ (hd } o \text{ ALL } f) :: \text{hd } o \text{ ALL } f = f \circ \text{hd}$$

$$\text{LALL} : \text{DEF } o \text{ (last } o \text{ ALL } f) :: \text{last } o \text{ ALL } f = f \circ \text{last}$$

$$\text{TLALL} : \text{DEF } o \text{ ALL } f :: t1 \circ \text{ALL } f == \text{ALL } f \circ t1$$

première partie

HALL, LALL et TLALL s'orientent de gauche à droite et créent chacun une paire critique avec E1 donnant les nouvelles règles .

HALL':  $hd\ o\ (ALL\ f\ o\ h) \Rightarrow f\ o\ (hd\ o\ h)$  SI  $(DEF\ o\ (ALL\ f\ o\ h))$

LALL':  $last\ o\ (ALL\ f\ o\ h) \Rightarrow f\ o\ (last\ o\ h)$  SI  $(DEF\ o\ (ALL\ f\ o\ h))$

TLALL':  $tl\ o\ (ALL\ f\ o\ h) \Rightarrow ALL\ f\ o\ (tl\ o\ h)$  SI  $(DEF\ o\ (ALL\ f\ o\ h))$

#### Introduction de ALLAPP

ALLAPP:  $append\ o\ [ALL\ f\ o\ a, ALL\ f\ o\ g] = ALL\ f\ o\ (append\ o\ [a, b])$

- On oriente ALLAPP de gauche à droite

Une superposition de ALLALL' à l'occurrence 21 donne la nouvelle règle

$append\ o\ [ALL(f\ o\ g)\ o\ a, ALL\ f\ o\ b] \Rightarrow ALL\ f\ o\ (append\ o\ [ALL\ g\ o\ a, b])$

La superposition de E1 à l'occurrence 211 engendre des règles

$append\ o\ [ALL(f1\ o\ (f2\ o\ \dots (fn\ o\ g)\ \dots))\ o\ a, ALL(f1\ o\ (f2\ o\ \dots fn)\ \dots)\ o\ b]$   
 $\Rightarrow ALL(f1\ o\ f2\ o\ \dots fn)\ \dots\ o\ (append\ o\ [ALL\ g\ o\ a, b])$

et REVE diverge.

- On oriente ALLAPP de droite à gauche

ALLAPP:  $ALL\ f\ o\ (append\ o\ [a, b]) \Rightarrow append\ o\ [ALL\ f\ o\ a, ALL\ f\ o\ b]$

et ce problème disparaît.

#### Introduction de ALLCONS

ALLCONS:  $cons\ o\ [f\ o\ x, ALL\ f\ o\ a] = ALL\ f\ o\ (cons\ o\ [x, a])$

est réduit par DEFCONS et orienté de droite à gauche en une règle

ALLCONS:  $ALL\ f\ o\ (append\ o\ [L(x), a]) \Rightarrow append\ o\ [L(f\ o\ x), ALL\ f\ o\ a]$

première partie

qui est réduite par ALLAPP en

ALLCONS:  $append\ o\ [ALL\ f\ o\ L(x), ALL\ f\ o\ a]$   
 $\Rightarrow append\ o\ [L(f\ o\ x), ALL\ f\ o\ a]$

qui fait diverger REVE (avec les superpositions de APPASS).

Mais un axiome sur "L":

ALLLIST:  $ALL\ f\ o\ L(x) = L(f\ o\ x)$

s'oriente de gauche à droite, détruit ALLCONS, ce qui fait disparaître le problème.

ALLLIST donne, par superposition sur TLALL et TLALL', des paires critiques réductibles à "!" que l'on "jette".

#### Introduction de ALLVID

ALLVID:  $ALL\ f\ o\ \langle \rangle^- = \langle \rangle^-$

ALLVID s'oriente de gauche à droite.

Toutes les paires critiques trouvées avec HALL, HALL', LALL, LALL', TLALL et TLALL' sont à rejeter comme étant équivalentes à "!".

#### Introduction de TDISTL

TDISTL:  $trans\ o\ [ALL?l\ o\ (distl\ o\ [x, a]), a] = distl\ o\ [x, a]$

TDISTL s'oriente de gauche à droite. La paire critique provoquée par APPDISTL est détruite par ALLAPP.

#### Introduction de ALLID et FALL

première partie

ALLID:  $\text{ALL } f \circ \text{id} = \text{id}$

FALL:  $\text{trans } o [\text{ALL } f, \text{ALL } g] = \text{ALL}[f, g]$

Pour diminuer le nombre de séquences intermédiaires, on oriente FALL de gauche à droite en une règle

FALL:  $\text{trans } o [\text{ALL } f, \text{ALL } g] \Rightarrow \text{ALL}[f, g]$

L'orientation contraire fait diverger REVE (à cause de la superposition de ALLALL' puis de E1).

Une paire critique est générée avec E1 donnant la règle

FALL':  $\text{trans } o [\text{ALL } f \circ h, \text{ALL } g \circ h] \Rightarrow \text{ALL}[f, g] \circ h$

Pour ces deux règles, des paires critiques sont obtenues avec ALLID

FALLg:  $\text{trans } o [\text{ALL } f, \text{id}] = \text{ALL}[f, \text{id}]$

FALLd:  $\text{trans } o [\text{id}, \text{ALL } f] = \text{ALL}[\text{id}, f]$

FALL'g:  $\text{trans } o [\text{ALL } f \circ h, h] = \text{ALL}[f, \text{id}] \circ h$

FALL'd:  $\text{trans } o [h, \text{ALL } f \circ h] = \text{ALL}[\text{id}, f] \circ h$

TALL:  $\text{ALL}[\text{id}, \text{id}] \circ h = \text{trans } o [h, h]$

On oriente les trois premières équations de gauche à droite et la dernière de droite à gauche.

Notons que TALL détruit la règle TVID de S-PRIMITIVE.

Avec une autre orientation, REVE diverge (à cause des superpositions de ALLALL puis de E1).

```

+++++
+ orientation manuelle: S-ALL
+ +++++
+
+ S-PRIMITIVE et
+
+ +++++
+
+ ALLALL: $\text{ALL } f \circ \text{ALL } g \Rightarrow \text{ALL}(f \circ g)$
+
+ ALLALL': $\text{ALL } f \circ (\text{ALL } g \circ h) \Rightarrow \text{ALL}(f \circ g) \circ h$
+
+ HALL: $\text{hd } o \text{ALL } f \Rightarrow f \circ \text{hd}$ SI (DEF o ALL f)
+
+ HALL': $\text{hd } o (\text{ALL } f \circ h) \Rightarrow f \circ (\text{hd } o h)$ SI (DEF o (ALL f o h))
+
+ LALL: $\text{last } o \text{ALL } f \Rightarrow f \circ \text{last}$ SI (DEF o ALL f)
+
+ LALL': $\text{last } o (\text{ALL } f \circ h) \Rightarrow f \circ (\text{last } o h)$ SI (DEF o (last o h))
+
+ TLALL: $\text{tl } o \text{ALL } f \Rightarrow \text{ALL } f \circ \text{tl}$ SI (DEF o ALL f)
+
+ TLALL': $\text{tl } o (\text{ALL } f \circ h) \Rightarrow \text{ALL } f \circ (\text{tl } o h)$
+
+ SI (DEF o (ALL f o h))
+
+ ALLID: $\text{ALL } \text{id} \Rightarrow \text{id}$
+
+ ALLLIST: $\text{ALL } f \circ L(x) \Rightarrow L(f \circ x)$
+
+ ALLAPP: $\text{ALL } f \circ (\text{append } o [a, b]) \Rightarrow \text{append } o [\text{ALL } f \circ a, \text{ALL } f \circ b]$
+
+ ALLVID: $\text{ALL } f \circ \langle \rangle^- \Rightarrow \langle \rangle^-$
+
+ FALL: $\text{trans } o [\text{ALL } f, \text{ALL } g] \Rightarrow \text{ALL}[f, g]$
+
+ FALL': $\text{trans } o [\text{ALL } f \circ h, \text{ALL } g \circ h] \Rightarrow \text{ALL}[f, g] \circ h$
+
+ FALLg: $\text{trans } o [\text{ALL } f, \text{id}] \Rightarrow \text{ALL}[f, \text{id}]$
+
+ FALLd: $\text{trans } o [\text{id}, \text{ALL } g] \Rightarrow \text{ALL}[\text{id}, g]$
+
+ FALL'g: $\text{trans } o [\text{ALL } f \circ h, h] \Rightarrow \text{ALL}[f, \text{id}] \circ h$
+
+ FALL'd: $\text{trans } o [h, \text{ALL } f \circ h] \Rightarrow \text{ALL}[\text{id}, f] \circ h$
+
+ TALL: $\text{trans } o [a, a] \Rightarrow \text{ALL}[\text{id}, \text{id}] \circ a$
+
+ TDISTL: $\text{trans } o [\text{ALL?1 } o (\text{distl } o [x, a]), a] \Rightarrow \text{distl } o [x, a]$
+
+++++

```

Tenter d'ajouter les axiomes ALL?iTRANS fait diverger REVE

ALL?1TRANS: ALL ?1 o (trans o [a,b]) => a SI (DEF o (trans o [a,b]))

ALL?2TRANS: ALL ?2 o (trans o [a,b]) => b SI (DEF o (trans o [a,b]))

Par exemple une paire critique est obtenue avec ALLALL' donnant une nouvelle règle

ALL(f o ?1) o (trans o [a,b]) => ALL f o a SI (DEF o (trans o [a,b]))

Mais REVE diverge ensuite avec les nouvelles règles obtenues par les superpositions des règles de S-BASE.

En particulier, avec E1, on obtient des nouvelles règles

ALL (f1 o (f2 o (f3 o ... (fn o ?1)...)) o (trans o [a,b]))

=> ALL(f1 o (f2 o (f3 o ... o fn)...)) o a SI (DEF o (trans o [a,b]))

Tenter d'ajouter les axiomes ALLDISTL et ALL+IOTA fait aussi diverger REVE

ALLDISTL: ALL[f o ?1, g o ?2] o (distl o [x,a])

=> distl o [f o x, ALL g o a]

ALL+IOTA: ALL(+ o [1<sup>-</sup>, id]) o iota => tl o (iota o (+ o [1<sup>-</sup>, id]))

Pour ALLDISTL les superpositions de E1 aux occurrences 111 et 112, par exemple, engendrent une famille infinie de règles nouvelles.

Pour ALL+IOTA, une nouvelle règle est obtenue par superposition de ALLALL'.

ALL+IOTA': ALL(f o (+ o [1<sup>-</sup>, id])) o iota

=> ALL f o (tl o (iota o (+ o [1<sup>-</sup>, id])))

et avec E1, superposée à l'occurrence 11, on obtient une famille infinie de nouvelles règles.

Nous étudierons ces divergences dans la deuxième partie  
 ~~~~~

première partie

### 3.4. Les axiomes sur "\*"

#### L'axiome \*ALL

Pour diminuer le nombre des séquences intermédiaires, on oriente \*ALL de gauche à droite.

\*ALL: \*f o [x, ALL g o h] => \*(f o [?1, g o ?2]) o [x, h]

#### \*ALL et S-BASE

L'orientation inverse fait diverger REVE à cause des superpositions de E1 à l'occurrence 1122.

La superposition de P2 donne une nouvelle règle

\*ALL': \*f o [x, ALL g] => \*(f o [?1, g o ?2]) o [x, id]

#### \*ALL et ALLID

La superposition de ALLID donne la nouvelle règle

\*(f o [?1, ?2]) o [x, a] => \*f o [x, a]

Cette règle fait diverger REVE à cause des superpositions de E1 à l'occurrence 11.

Nous remarquons que l'équation correspondante indique simplement que f dans \*f est d'arité 2. L'axiome suivant

\*(f o [?1, ?2]) = \*f

détruit la nouvelle règle mais REVE diverge avec les superpositions de E1.

Première partie

### Introduction du nouvel axiome ID

Nous n'utilisons pour le moment que l'opérateur "[\_]" d'arité 2 et en ce cas l'axiome

ID: DEF o [?1,?2] :: [?1,?2] = id  
 ~~~~~

est valide. Il détruit la paire critique

$*(f \circ [?1,?2]) \circ [x,a] = *f \circ [x,a]$

Alors il n'y a plus de problème entre \*ALL et ALLID.

Notons que pour que cet axiome soit généralisable, il suffira de  
 ~~~~~

distinguer les sélecteurs selon leur arité  
 ~~~~~

(Nous le ferons dans la deuxième partie).

ID s'oriente de gauche à droite

ID: [?1,?2] => id SI (DEF o [?1,?2])

ID se superpose avec E2 donnant une nouvelle règle

ID': [?1 o f, ?2 o f] => f SI (DEF o [?1 o f, ?2 o f])

ID et ID' complètent S-BASE en un système convergent S-BASE'.

première partie

```

+++++
+ +
+ S-BASE' +
+ ***** +
+ S-BASE et +
+ ***** +
+ ID: [?1,?2] => id SI (DEF o [?1,?2]) +
+ ID': [?1 o f, ?2 o f] => f SI (DEF o [?1 o f, ?2 o f]) +
+ ***** +
+++++

```

ID et ID' se superposent sur les règles qui contiennent en partie gauche [x,y]. Par exemple avec LTRANS, on obtient deux nouvelles règles:

LTRANS': last o trans => [last o ?1, last o ?2] SI (DEF o trans)

LTRANS'': last o (trans o x)

=> [last o (?1 o x), last o (?2 o x)] SI (DEF o (trans o x))

LTRANS'' détruit LTRANS.

Cela ne peut aller que si l'orientation n'est pas modifiée. Mais par exemple avec la règle TLTRANS, on obtient la paire critique

TLTRANS': t1 o trans = trans o [t1 o ?1, t1 o ?2]

que REVE oriente automatiquement (même si toutes les règles sont orientées "manuellement") de droite à gauche, ce qui provoque une divergence.

De la même manière, REVE ne donne pas l'orientation voulue pour les nouvelles règles issues de ID ou ID' et des règles TLTRANS, TLDISTL, TLAPP, APPASS, ALLAPP et aussi TL\*

TL\*: t1 o (\*f o [x,a]) = \*f o [f o [x,hd o a], t1 o a] SI (DEF o (t1 o a))  
 ~~~~~

TL\* est orientée de droite à gauche, REVE diverge avec, par exemple, les

première partie

superpositions de E1 à l'occurrence 21.

Nous étudierons plus loin les problèmes soulevés par TL\* et ALLAPP.

Nous savons résoudre le problème pour les axiomes TLTRANS, TLDISTL et TLAPP, il suffit de remplacer "trans", "distl" et "append" par des opérateurs "T", "D" et "A" tels que

DEFT:  $\text{trans } o [a,b] = T(a,b)$

DEFD:  $\text{distl } o [x,b] = D(x,b)$

DEFA:  $\text{append } o [a,b] = A(a,b)$

qui avec les superpositions de E1, se complètent avec

TRANS:  $T(a,b) o h \Rightarrow T(a o h, b o h)$

DISTL:  $D(x,b) o h \Rightarrow D(a o h, b o h)$

APP:  $A(a,b) o h \Rightarrow A(a o h, b o h)$

Les règles sur les primitives sont réécrites. Par exemple

LTRANS:  $\text{last } o (\text{trans } o [a,b]) \Rightarrow [\text{last } o a, \text{last } o b]$

SI (DEF o (trans o [a,b])

devient

LTRANS:  $\text{last } o T(a,b) \Rightarrow [\text{last } o a, \text{last } o b]$  SI (DEF o T(a,b))

On évite alors les superpositions avec ID et ID' des règles ainsi réécrites et, par conséquent, les nouvelles règles comme LTRANS' et LTRANS".

Notons que si nous introduisons les règles DEFT, DEFD et DEFA, nous obtenons par superposition de ID les nouvelles équations suivantes

première partie

DEFT':  $\text{trans} = T(?1, ?2)$

DEFD':  $\text{distl} = D(?1, ?2)$

DEFA':  $\text{append} = A(?1, ?2)$

DEFT', DEFD' et DEFA' détruisent DEFT, DEFD et DEFA.

Considérons aussi

DEFCONS:  $\text{cons } o [x,a] \Rightarrow \text{append } o [L(x), a]$ .

DEFCONS est alors réécrit en

DEFCONS:  $\text{cons } o [x,a] \Rightarrow A(L(x), a)$

La superposition de ID donne la nouvelle équation

DEFCONS':  $\text{cons} = A(L(?1), ?2)$  qui détruit DEFCONS.

Nous définissons un morphisme MM des FP-termes dans les OP-termes  $T(S- \{ \text{trans}, \text{distl}, \text{append}, \text{cons} \} + \{ T, D, A, L \}, V)$  grace aux définitions DEFT', DEFD', DEFA' et DEFCONS'.

Pour les OP-termes, on peut orienter convenablement les axiomes issus de TLTRANS, TLDISTL et TLAPP. REVE se termine et on obtient un système S-PRIMITIVE'.

REVE ne nous autorise pas, même manuellement, à obtenir le système S-PRIMITIVE' équivalent (au sens des chemins de réécriture) pour les FP-termes. C'est pourquoi nous continuons avec le système pour les OP-termes.

première partie

```

+*****+
+ o{[],id,l,-,L,T,D,A}
+ t1>[T,D,A], T>[L,[],A], D>[L,[],A]
+ S-PRIMITIVE'
+*****+
+ S-BASE'
+*****+
+ LTL: last o t1 => last SI (DEF o (last o t1))
+
+ LTL': last o (t1 o h) => last o h SI (DEF o (last o (t1 o h)))
+
+ LIST: L(x) o y => L(x o y)
+
+ HLIST: hd o L(x) => x
+
+ LLIST: last o L(x) => x
+
+ TLLIST: t1 o L(x) => <>~ SI (DEF o x)
+
+ TRANS: T(a,b) o h => T(a o h,b o h)
+
+ HTRANS: hd o T(a,b) => [hd o a,hd o b] SI (DEF o T(a,b))
+
+ LTRANS: last o T(a,b) => [last o a,last o b] SI (DEF o T(a,b))
+
+ TLTRANS: t1 o T(a,b) => T(t1 o a,t1 o b)
+
+ TLIST: T(L(x),L(y)) => L([x,y])
+
+ TVID: T(<>~,<>~) => <>~
+
+ DISTL: D(x,a) o h => D(x o h,a o h)
+
+ HDISTL: hd o D(x,a) => [x,hd o a]
+
+ LDISTL: last o D(x,a) => [x,last o a]
+
+ TLDISTL: t1 o D(x,a) => D(x,t1 o a)
+
+ DLIST: D(x,L(y)) => L([x,y])
+
+ DVID: D(x,<>~) => <>~ SI (DEF o x)
+
+ HIOTA: hd o iota => 1~ SI (DEF o iota)
+
+ HIOTA': hd o (iota o n) => 1~ SI (DEF o (iota o n))
+
+ LIOTA: last o iota => id SI (DEF o iota)
+
+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))

```

première partie

```

+
+ APP: A(a,b) o h => A(a o h,b o h)
+
+ HAPP: hd o A(a,b) => hd o a SI ((DEF o A(a,b)) & (DEF o (hd o a)))
+
+
+ LAPP: last o A(a,b) => last o b
+
+
+ SI ((DEF o A(a,b)) & (DEF o (last o b)))
+
+
+ TLAPP: t1 o A(a,b) => A(t1 o a,b) SI (DEF o (t1 o a))
+
+ APPASS: A(A(a,b),c) => A(a,A(b,c))
+
+ APPDISTL: D(x,A(a,b)) => A(D(x,a),D(x,b))
+
+ APPTRANS: T(A(a,b),A(c,d)) => A(T(a,c),T(b,d))
+
+
+ SI (DEF o (A(T(a,c),T(b,d))))
+
+
+ APPVID: A(a,<>~) => a SI (DEF o A(a,<>~))
+
+ APPVIDS: A(<>~,a) => a SI (DEF o A(<>~,a))
+
+*****+

```

Nous étudions maintenant les superpositions entre \*ALL et les autres règles de S-ALL

\*ALL' et les règles de S-ALL

Une paire critique est obtenue avec ALLVID

$$*f o [x, <>^{\sim}] = *(f o [?1, g o ?2]) o [x, <>^{\sim}]$$

qui est détruite par \*VID.

$$*VID: *f o [x, <>^{\sim}] = L(x)$$

Il y a aussi une superposition de ALLLIST sur ALLALL' donnant la nouvelle règle

première partie

$$*f \circ [x, L(y \circ z)] \Rightarrow *(f \circ [?1, y \circ ?2]) \circ [x, L(z)]$$

Puis REVE diverge (à cause des superpositions à l'occurrence 221 de E1 par exemple). Cette nouvelle règle peut être détruite par l'introduction d'un axiome

$$*LIST: *f \circ [x, L(y)] = \text{append} \circ [L(x), L(f \circ [x, y])]$$

~~~~~

ce qui résout le problème.

Il se produit alors une paire critique entre ALLAPP et \*ALL'

$$*f \circ [x, A(ALL \ g \circ a, ALL \ g \circ b)] = *(f \circ [?1, g \circ ?2]) \circ [x, A(a, b)]$$

Si nous l'ordonnons de droite à gauche, REVE diverge (à cause des superpositions de E1 à l'occurrence 122). Si on l'oriente de gauche à droite, REVE diverge encore (car des nouvelles règles sont issues de superpositions de ALLALL et provoquent des divergences avec les superpositions de E1)

Cependant, cette paire critique est réduite par l'axiome \*APP

$$*APP: *f \circ [x, A(a, b)] = A(*f \circ [x, a], t1 \circ (*f \circ [last \circ (*f \circ [x, a]), b]))$$

qu'on oriente de gauche à droite.

Mais APPVID provoque une paire critique sur \*APP

$$*f \circ [x, b] = A(L(x), t1 \circ (*f \circ [x, b]))$$

qui s'oriente de droite à gauche et provoque une divergence (par exemple à cause des superpositions de APPASS). Nous remarquons que cette paire critique est détruite par l'axiome CONS que nous n'avons pu introduire dans

première partie

S-PRIMITIVE.

Nous ne pouvons pas avoir ni ALLAPP, ni \*APP avec \*ALL. Remarquons que cela nous empêche d'avoir TDISTL (à cause des paires critiques issues de la superposition entre TDISTL et APPDISTL qui ne sont plus détruites par ALLAPP)

Pour poursuivre, nous isolons un système de réécriture S-ALL' construit avec S-PRIMITIVE' et les axiomes de S-ALL à l'exception de ALLAPP et de TDISTL.

Première partie

```

+++++
+ orientation manuelle: S-ALL'
+ *****
+ S-PRIMITIVE'
+ *****
+ ALLALL: ALL f o ALL g => ALL(f o g)
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f)
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h))
+ LALL: last o ALL f => f o last SI (DEF o ALL f)
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (ALL f o h))
+ TLALL: tl o ALL f => ALL f o tl SI (DEF o ALL f)
+ TLALL': tl o (ALL f o h) => ALL f o (tl o h) SI (DEF o (ALL f o h))
+ ALLID: ALL id => id
+ ALLLIST: ALL f o L(x) => L(f o x)
+ ALLVID: ALL f o <>^- => <>^-
+ FALL: T(ALL f, ALL g) => ALL[f, g]
+ FALL': T(ALL f o h, ALL g o h) => ALL[f, g] o h
+ FALLg: T(ALL f, id) => ALL[f, id]
+ FALLd: T(id, ALL f) => ALL[id, f]
+ FALL'g: T(ALL f o h, h) => ALL[f, id] o h
+ FALL'd: T(h, ALL f o h) => ALL[id, f] o h
+ TALL: T(a, a) => ALL[id, id] o a
+++++

```

Introduisons maintenant les autres axiomes sur "x"

- On peut introduire H\*

première partie

$H^*: hd\ o\ (*f\ o\ [x,a]) = x\ SI\ (DEF\ o\ (*f\ o\ [x,a]))$

Avec ID et ID' on obtient

$H^{*'}: hd\ o\ *f = ?1\ SI\ (DEF\ o\ (hd\ o\ *f)),$

$H^{*''}: hd\ o\ (*f\ o\ x) = ?1\ o\ x\ SI\ (DEF\ o\ (hd\ o\ (*f\ o\ x)))$

H\* est détruite par H\*''.

- On peut introduire \*CONS

$*CONS: *f\ o\ [x, cons\ o\ [y,a]] = cons\ o\ [x, *f\ o\ [f\ o\ [x,y],a]]$

qui s'exprime dans les OP-termes en

$*CONS: *f\ o\ [x, A(L(y),a)] = A(L(x), *f\ o\ [f\ o\ [x,y],a])$

et s'oriente de gauche à droite.

- On peut introduire TL\*\*IOTA

$TL^{**}IOTA: tl\ o\ (*(+ o\ [?1,1^-]) o\ [0^-,iota]) = iota$

qui s'oriente de gauche à droite. La superposition de E1 donne une nouvelle règle

$TL^{**}IOTA': tl\ o\ (*(+ o\ [?1,1^-]) o\ [0^-,iota\ o\ n]) = iota\ o\ n$

La superposition sur LTL' donne les nouvelles règles

$L^{**}IOTA: last\ o\ (*(+ o\ [?1,1^-]) o\ [0^-,iota]) = id$

$L^{**}IOTA: last\ o\ (*(+ o\ [?1,1^-]) o\ [0^-,iota\ o\ n]) = n$

Le problème F\* et TALL

première partie

F\*: trans o [\*f o [x,a],\*g o [y,a]]  
 = [\*f o [?1o?1,?2],g o [?2o?1,?2]] o [[x,y],a]

La superposition de TALL

TALL: trans o [a,a] => ALL[id,id] o a

donne une paire critique

P1: [\*f o [?1o?1,?2],f o [?2o?1,?2]] o [[x,x],a]  
 = ALL[id,id] o [\*f o [x,a]]

L'orientation dans le sens gauche droite fait diverger REVE (par exemple avec les superpositions de E1 aux occurrences 111 et 112).

L'orientation inverse donne des paires critiques complexes issues des superpositions avec ALLALL et \*ALL. L'on peut constater que REVE diverge en engendrant des règles comme

ALL[ id , [id,id] ] o [\*f o [x,a]] =>  
 [\* f o [?1o?1,?2] , [ f o [?1o(?2o?1),?2],f o [?2o(?2o?1),?2] ] ]  
 o [[x,[x,x]],a]

Nous construirons ensuite un système de réécriture qui n'inclut pas ALLIO pour disposer de l'axiome F\*.

Nous pouvons cependant ajouter H\*, \*VID, \*LIST, \*CONS et \*ALL à S-ALL'. REVE se termine. Nous appelons ce système S-\*

première partie

```

+++++
+
+ orientation manuelle: S-*
+
+ *****
+ S-ALL'
+
+ *****
+
+ H*: hd o *f => ?1 SI (DEF o *f)
+
+ H*: hd o (*f o x) => ?1 o x SI (DEF o (*f o x))
+
+ *VID: *f o [x,<>] => L(x)
+
+ *LIST: *f o [x,L(y)] => A(L(x),L(f o [x,y]))
+
+ *CONS: *f o [x,A(L(y),a)] => A(L(x),*f o [f o [x,y],a])
+
+ TL*+IOTA: t1 o (*(+ o [?1,1^-]) o [0^-,iota]) => iota
+
+ TL*+IOTA': t1 o (*(+ o [?1,1^-]) o [0^-,iota o n]) => iota o n
+
+ L*+IOTA: last o (*(+ o [?1,1^-]) o [0^-,iota]) => id
+
+ L*+IOTA': last o (*(+ o [?1,1^-]) o [0^-,iota o n]) => n
+
+ *ALL: *f o [x,ALL g o a] => *(f o [?1,g o ?2]) o [x,a]
+
+ *ALL': *f o [x,ALL g] => *(f o [?1,g o ?2]) o [x,id]
+
+++++

```

#### Problèmes posés par les autres axiomes

##### ● L'axiome TL\*

orienté de gauche à droite donne la règle:

TL\*: t1 o [\*f o [x,a]] => \*f o [f o [x,hd o a],t1 o a] SI (DEF o (t1 o a))

(l'orientation inverse provoque une divergence avec les superpositions de E1 à l'occurrence 21). Nous essayons d'ajouter cette règle à S-PRIMITIVE.

première partie

Une superposition se produit avec LTL' donnant la paire critique

$$\text{last } o (*f \ o \ [x,a]) = \text{last } o (*f \ o \ [f \ o \ [x,hd \ o \ a],tl \ o \ a])$$

La réécriture est infinie avec une orientation de gauche à droite. REVE diverge si on l'oriente de droite à gauche (par exemple avec les superpositions de E1 à l'occurrence 221).

TL\* réécrit TL\*\*IOTA (et TL\*\*IOTA')

$$\text{TL**IOTA: } *(+ \ o \ [?1,1^-]) \ o \ [+ \ o \ [0^-,1^-],tl \ o \ \text{iota}] \Rightarrow \text{iota}$$

Alors, la superposition de H\*' donne l'équation

$$+ \ o \ [0^-,1^-] = 1^-$$

qui serait réduite par l'axiome PLUSO (0 élément neutre de +).

Remarquons que TL\* réécrit \*APP mais produit aussi une paire critique

$$A(*f \ o \ [f \ o \ [x,hd \ o \ a],tl \ o \ a]),$$

$$*f \ o \ [f \ o \ [\text{last } o (*f \ o \ [x,a]),hd \ o \ b]),tl \ o \ b])$$

$$= A(*f \ o \ [f \ o \ [x,hd \ o \ a],tl \ o \ a],$$

$$*f \ o \ [f \ o \ [\text{last } o (*f \ o \ [f \ o \ [x,hd \ o \ a],tl \ o \ a]),hd \ o \ b],tl \ o \ b])$$

qui n'est réduite que par la paire critique obtenue entre TL\* et LTL (qui provoque une divergence).

● L'axiome \*?2

$$*?2: \ *?2 = \text{cons} = A(L(?1),?2)$$

On oriente de gauche à droite. Une paire critique avec \*ALL s'écrit

première partie

$$A(L(x),ALL \ g \ o \ a) = *(?2 \ o \ g) \ o \ [x,a]$$

On l'oriente de droite à gauche et REVE diverge (par exemple avec les superpositions de E1).

● L'axiome TL\*?1

$$\text{TL*?1: } tl \ o \ *?1 = ALL?1 \ o \ \text{dist1}$$

s'oriente de gauche à droite.

$$\text{TL*?1: } tl \ o \ *?1 \Rightarrow ALL?1 \ o \ D(?1,?2)$$

Une nouvelle règle est engendrée par superposition avec E1

$$\text{TL*?1': } tl \ o \ (*?1 \ o \ z) \Rightarrow ALL?1 \ o \ D(?1 \ o \ z,?2 \ o \ z)$$

La superposition avec LTL donne

$$L*?1: \text{last } o \ *?1 \Rightarrow ?1 \text{ et } L?1': \text{last } o \ (*?1 \ o \ z) \Rightarrow ?1 \ o \ z$$

La superposition de \*ALL donne

$$ALL?1 \ o \ D(x,id) = ALL?1 \ o \ D(x,ALL \ g)$$

qui s'oriente de droite à gauche mais alors la superposition de ALLALL' puis de E1 fait diverger REVE.

La superposition de TL\* donne

$$*?1: \ *?1 \ o \ [x,tl \ o \ a] \Rightarrow ALL?1 \ o \ D(x,a)$$

Mais alors TL\* se superpose à l'occurrence 22 et REVE diverge.

● L'axiome ALL\*IOTA

première partie

ALL\*IOTA: ALL(last o (\*f o [g o x<sup>-</sup>,iota])) o iota  
 = tl o (\*f o [g o x<sup>-</sup>,iota])

REVE diverge (par exemple avec les superpositions de E1 à l'occurrence 11221).

● L'axiome APPIOTA

APPIOTA: A(iota o p, tl o (\*(+ o ([?1,1<sup>-</sup>]) o [p,iota o (+ o [1<sup>-</sup>, - o [n,p]])]))  
 => iota o n SI ((DEF o (~ o [n,p]) & (DEF o (iota o p)))

REVE diverge (avec les superpositions de APPASS).

● L'axiome F\*

Nous avons vu que F\* ne pouvait pas s'introduire avec TALL nous l'essayons donc sans la présence de l'axiome ALLID dont est issue TALL.

Signalons aussi que la paire critique produite avec TLTRANS n'est réduite que par TL\*.

Une paire critique est obtenue avec LTRANS:

FL\*: {last o (\*f o [x,a]), last o (\*g o [y,a])} =  
 last o (\*[f o [?1o?1,?2], g o [?2o?1,?2]] o [[x,y],a])

Si on l'oriente de droite à gauche, REVE diverge (avec les superpositions de E1 aux occurrences 211 et 212). Si on l'oriente de gauche à droite, on a superposition de P1 (ou P2) donnant

L\*?1: last o (\*f o [x,a]) =

?1 o (last o (\*[f o [?1o?1,?2], g o [?2o?1,?2]] o [[x,y],a])

qui s'oriente de droite à gauche et fait diverger REVE (à cause des super-

positions de E1 aux occurrences 2211 et 2212 par exemple).

● Les axiomes FALL\*, F\*\*

Avec LTRANS, on a la même chose que pour F\*.

TL\* réécrit le membre droit de F\*\* et REVE diverge comme pour TL?1 (voir le paragraphe sur l'axiome TL?1).

Nous cherchons à obtenir un système de réécriture contenant les axiomes F\*, FALL\* et F\*\*.

Un système contenant les axiomes F\*, FALL\* et F\*\*

Pour avoir F\*, il ne faut pas TALL donc on ne peut pas avoir ALLID.

Pour avoir F\*\*, il ne faut pas TL\* et on ne peut donc pas avoir non plus TLTRANS. Pour avoir F\*, FALL\* et F\*\*, il ne faut pas avoir LTRANS.

Nous reprenons S-PRIMITIVE sans LTRANS et TLTRANS. REVE se termine dans les mêmes conditions que pour S-PRIMITIVE. On appelle S-PRIM1 ce système.

Nous ajoutons les axiomes de ALL suivants: ALLALL qui donne ALLALL', FALL qui donne FALL', HALL qui donne HALL', LALL qui donne LALL' et TALL qui donne TALL'. REVE se termine donnant un système de réécriture S-ALL1.

```

+++++
+ ordonnancement manuel: S-ALL1
+ *****
+ S-PRIM1 S-PRIMITIVE sans LTRANS et TLTRANS
+ *****
+
+ ALLALL: ALL f o ALL g => ALL(f o g)
+
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h
+
+ FALL: trans o [ALL f, ALL g] => ALL[f, g]
+
+ FALL': trans o [ALL f o h, ALL g o h] => ALL[f, g] o h
+
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f)
+
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h))
+
+ LALL: last o ALL f => f o last SI (DEF o ALL f)
+
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (ALL f o h))
+
+ TLALL: t1 o ALL f => ALL f o t1 SI (DEF o ALL f)
+
+ TLALL': t1 o (ALL f o h) => ALL f o t1 SI (DEF o (ALL f o h))
+
+++++

```

On peut alors ajouter les axiomes de "\*" suivants: \*ALL qui donne \*ALL', H\*, F\*, FALL\*, FALL\*S, F\*\*, F\*\*S. On obtient un système de réécriture localement confluent S-\*\*.

```

+++++
+ ordonnancement manuel: S-**
+ *****
+ S-ALL1
+ *****
+
+ *ALL: *f o [x, ALL g o h] => *(f o [?1, g o ?2]) o [x, h]
+
+ *ALL': *f o [x, ALL g] => *(f o [?1, g o ?2]) o [x, id]
+
+ H*: hd o (*f o [x, a]) => x SI (DEF o (*f o [x, a]))
+
+ F*: trans o [*f o [x, a], *g o [y, a]]
+
+ => *f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a]
+
+ FALL*: trans o [ALL g o (*f o [x, a]), *f o [x, a]]
+
+ => *[g o (f o [?2o?1, ?2]), f o [?2o?1, ?2]] o [[g o x, x], a]
+
+ FALL*S: trans o [*f o [x, a], ALL g o (*f o [x, a])]
+
+ => *f o [?1o?1, ?2], g o (f o [?1o?1, ?2])] o [[x, g o x], a]
+
+ F**: trans o [*f o [x, t1 o (*g o [y, a])], *g o [y, a]]
+
+ => *f o [?1o?1, g o [?2o?1, ?2]], g o [?2o?1, ?2]] o [[x, y], a]
+
+ F**S: trans o [*g o [y, a], *f o [x, t1 o (*g o [y, a])]]
+
+ => *[g o [?1o?1, ?2], f o [?2o?1, g o [?1o?1, ?2]]] o [[y, x], a]
+
+++++

```

Nous ne pouvons pas ajouter \*VID et \*LIST à S-\*\*

A cause de ALLALL', ALLVID, \*VID, \*LIST et ALLLIST sont inséparables.

Cela irait avec F\*.

Avec F\*\*, la superposition de \*LIST donne la paire critique:

trans o [\*f o [x, A(<>, L(g o [y, z]))], A(L(y), L(g o [y, z]))]  
 = A(L([x, y]), [f o [x, g o [y, z]], g o [y, z]])

qui est détruite par APPVID.

Avec FALL\* , la superposition de \*LIST donne la paire critique:

trans o [ALL g o A(L(x), L(f o [x, y])), A(L(x), L(f o [x, y]))]  
 = A(L(g o x, x), L(g o (f o [x, y]), f o [x, y]))

qui est détruite avec ALLAPP.

ALLALL' oblige à introduire alors \*APP (il ne faut donc plus APPVID).

Mais \*APP provoque une paire critique avec F\*

A(\*[f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a],  
 trans o [t1 o (\*f o [last o (\*f o [x, a]), b])),  
 t1 o (\*g o [last o (\*g o [y, a]), b]))]  
 = A(\*[f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a],  
 t1 o (\*[f o [?1o?1, ?2], g o [?2o?1, ?2]] o  
 [last o (\*[f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a]), b]))

qui ne peut être détruite que par TL\* (Le même problème se pose avec FALL\* et F\*\*).

On peut prendre directement \*APP réécrite par TL\*.

Nommons k le sous terme "[f o [?1o?1, ?2], g o [?2o?1, ?2]]". On trouve la paire critique

A(\*k o [[x, y], a],  
 \*k o [[f o [?1 o (last o (\*k o [[x, y], a])), hd o b],  
 g o [?2 o (last o (\*k o [[x, y], a])), hd o b]], t1 o b])  
 = A(\*k o [[x, y], a],  
 \*k o [[f o [last o (\*f o [x, a]), hd o b],  
 g o [last o (\*g o [y, a]), hd o b]], t1 o b])

qui est réduite par les paires critiques L\*?1 et L\*?2 issues de la superposition entre LTRANS et F\*. Mais nous avons vu que L?1 et L?2 faisaient diverger REVE.

Nous n'irons pas plus loin dans cette voie.

### 3.5. Les axiomes sur "WH"

Nous allons retrouver les problèmes que nous avons rencontrés pour introduire les axiomes de "\*".

#### ● L'axiome WHALL

Nous retrouvons des résultats semblables à ceux de \*ALL.

WHALL: WH(p, f) o [x, ALL g o h]  
 = WH(p o [?1, g o ?2], f o [?1, g o ?2]) o [x, h]

est orienté de gauche à droite afin de diminuer le nombre de séquences intermédiaires.

La superposition de P2 donne la nouvelle règle

$$\begin{aligned} \text{WHALL}': \text{WH}(p, f) \circ [x, \text{ALL } g] \\ = \text{WH}(p \circ [?1, g \circ ?2], f \circ [?1, g \circ ?2]) \circ [x, \text{id}] \end{aligned}$$

Il faut encore introduire ID (donc ID') et les opérateurs "A", "D", et "T" pour pouvoir inclure ALLID.

La paire critique avec ALLVID est détruite par WHVID.

WHVID:  $WH(p, f) \circ [x, \langle \rangle^-] = L(x)$

Avec ALLIST, il faut introduire les axiomes WHLIST1 et WHLIST2 correspondants aux deux préconditions complémentaires " $p \text{ o } [x,y]$ " et " $\text{not o } (p \text{ o } [x,y])$ " qu'il faut valider lors de la réécriture.

$$\text{WHLIST1: not } o(p \circ [x, y]) :: \text{WH}(p, f) \circ [x, L(y)] = L(x)$$
$$\text{WHLIST2: } p \circ [x, y] :: \text{WH}(p, f) \circ [x, L(y)] = A(L(x), L(f \circ [x, y]))$$

Ces axiomes servent à détruire la paire critique issue de la superposition entre ALLLIST et WHALL.

Remarquons qu'ils doivent être essayés séparément.

- On peut introduire HWH

$$\text{HWH: } \text{hd } o \text{ (WH(p,f) } o \text{ [x,a])} \Rightarrow x \text{ SI (DEF } o \text{ (WH(p,f) } o \text{ [x,a]))}$$

- On peut introduire WHCONS

$$\begin{aligned} \text{WHCONS: } p \circ [x, y] &:: \text{WH}(p, f) \circ [x, \text{cons} \circ [y, a]] \\ &= \text{cons} \circ [x, \text{WH}(p, f) \circ [f \circ [x, y], a]] \end{aligned}$$

qui s'oriente de gauche à droite et est réécrit par DEFCONS en

première partie

$$\Rightarrow A(L(x), WH(p, f) \circ [f \circ [x, y], a]) \text{ SI } (p \circ [x, y])$$

En complétant S-\* avec les axiomes WHALL, WHVID, WHLIST1 (et WHLIST2) HWH et WHCONS, REVE se termine et on obtient un système de réécriture S-WH. Cependant, il faut prouver que la précondition est valide avant de réécrire avec WHLIST1 (et WHLIST2) et WHCONS

```

+++++
+ orientation manuelle: S-WH
+ *****
+
+ S-*
+
+ *****
+
+ WHALL: WH(p,f) o [x,ALL g o h]
+
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,h]
+
+ WHALL': WH(p,f) o [x,ALL g]
+
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,id]
+
+ WHVID: WH(p,f) o [x,<~] => L(x)
+
+ WHLIST1: WH(p,f) o [x,L(y)] => L(x) SI (not o (p o [x,y]))
+
+
+ WHLIST2: WH(p,f) o [x,L(y)] => A(L(x),L(f o [x,y])) SI (p o [x,y])
+
+
+ WHCONS: WH(p,f) o [x,A(L(y),h)] =>
+
+ A(L(x),WH(p,f) o [f o [x,y],h]) SI (p o [x,y])
+
+
+ HWH: hd o WH(p,f) => ?1 SI (DEF o WH(p,f))
+
+ HWH': hd o (WH(p,f) o z) => ?1 o z SI (DEF o (WH(p,f) o z))
+
+++++

```

Nous envisageons maintenant l'introduction des autres axiomes

première partie

● L'axiome WHPART

WHPART: DEF  $o \ (WH(p \ o \ ?2, ?2) \ o \ [y, a]) ::$

$*f \ o \ [x, t1 \ o \ (WH(p \ o \ ?2), ?2) \ o \ [y, a]] = WH(p \ o \ ?2, f) \ o \ [x, a]$

Que cet axiome soit orienté dans un sens ou dans l'autre, réduit ou non par TLWH, la présence à gauche et à droite de " $p \ o \ ?2$ " provoque la divergence de REVE (à cause des superpositions de E1).

● L'axiome TLWH

TLWH:  $(p \ o \ [x, hd \ o \ a]) ::$

$t1 \ o \ (WH(p, f) \ o \ [x, a]) = WH(p, f) \ o \ [f \ o \ [x, hd \ o \ a], t1 \ o \ a]$

L'application de cet axiome nécessite toujours une preuve pour vérifier la précondition " $p \ o \ [x, hd \ o \ a]$ ".

TLWH pose avec LTL' le même problème que TL\*.

● L'axiome ALLAPP

ALLAPP:  $ALL \ f \ o \ A(a, b) \Rightarrow A(ALL \ f \ o \ a, ALL \ f \ o \ b)$

Il se superpose sur WHALL donnant la paire critique

$WH(p, f) \ o \ [x, A(ALL \ g \ o \ a, ALL \ g \ o \ b)] =$

$WH(p \ o \ [?1, g \ o \ ?2], f \ o \ [?1, g \ o \ ?2]) \ o \ [x, A(a, b)]$

Pour "WH", remarquons que l'on n'a pas d'axiome analogue à \*APP. Cette paire critique n'est pas réduite.

Si nous orientons cette équation de gauche à droite, REVE diverge avec les superpositions de ALLALL puis de E1.

première partie

● Les axiomes WH\* (et WH\*S) et WHWH (et WHWHS)

WH\*:  $WH(p, f) \ o \ [x, t1 \ o \ (*g \ o \ [y, a])]$

$= ALL?1 \ o \ (WH(p \ o \ [?1o?1, g \ o \ [?2o?1, ?2],$

$[f \ o \ [?1o?1, g \ o \ [?2o?1, ?2]], g \ o \ [?2o?1, ?2]) \ o \ [[x, y], a])$

WHWH:  $WH(p, f) \ o \ [x, t1 \ o \ (WH(q, g) \ o \ [y, a])] = ALL?1 \ o$

$(WH((q \ o \ [?2o?1, ?2]) \rightarrow ((p \ o \ [?1o?1, g \ o \ [?2o?1, ?2]) \rightarrow T^-; F^-); F^-$

$, [f \ o \ [?1o?1, g \ o \ [?2o?1, ?2]], g \ o \ [?2o?1, ?2]) \ o \ [[x, y], a])$

Dans le cas où la précondition de TLWH est vérifiée, TLWH réduit le membre gauche de WHWH. TL\* réduit le membre gauche de WH\*. Alors REVE diverge (à cause des superpositions de E1).

Avec WH\*, on obtient une paire critique issue de la superposition de \*LIST.

$WH(p, f) \ o \ [x, L(g \ o \ [y, z])] = ALL?1 \ o \ (WH(p \ o \ [?1o?1, g \ o \ [?2o?1, ?2]],$

$[f \ o \ [?1o?1, g \ o \ [?2o?1, ?2]], g \ o \ [?2o?1, ?2]) \ o \ [[x, y], L(z)])$

qui est réduite, dans le contexte " $\text{not } o \ (p \ o \ [x, f \ o \ [x, g \ o \ [y, z]])$ ", par WHLIST2 en

$A(L(x), L(f \ o \ [x, g \ o \ [y, z]]))$

$= ALL?1 \ o \ A(L([x, y]), L([f \ o \ [x, g \ o \ [y, z]], g \ o \ [y, z])))$

qui ne peut être réduite que par ALLAPP. Or, nous avons vu que ALLAPP et \*APP faisaient diverger REVE.

Le problème est le même avec WHWH et la superposition de WHLIST2.

● Les axiomes ALLWH (et ALLWHS) et \*WH (et \*WHS)

première partie

ALLWH: trans o [ALL g o (WH(p,f) o [x,a]),WH(p,f) o [x,a]]  
 = WH(p o [?2o?1,?2],[g o (f o [?2o?1,?2]),f o [?2o?1,?2]]) o [[g o x,x],a]  
 \*WH: trans o [\*f o [x,t1 o (WH(p,f) o [y,a]),WH(p,f) o [y,a]]  
 = WH(p o [?2o?1,?2],[f o [?1o?1,g o [?2o?1,?2]],g o [?2o?1,?2]]) o [[x,y],a]

Ils posent les mêmes problèmes que F\* avec LTRANS. En outre \*WH pose le même problème que WH\* avec TLWH. On peut par contre ajouter ces quatre axiomes accompagnés de WHALL et HWH à S-\*\* et l'on obtient un système de réécriture S-WHWH.

```
+++++
+ ordonnancement manuel: S-WHWH
+ *****
+
+ S-**
+
+ *****
+
+ WHALL: WH(p,f) o [x,ALL g o a]
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,a]
+
+ WHALL': WH(p,f) o [x,ALL g] => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,id]
+
+ HWH: hd o (WH(p,f) o [x,a]) => x SI (DEF o (WH(p,f) o [x,a]))
+
+ ALLWH: trans o [ALL g o (WH(p,f) o [x,a]),WH(p,f) o [x,a]]
+
+ => WH(p o [?2o?1,?2],[g o (f o [?2o?1,?2]),f o [?2o?1,?2]]) o [[g o x,x],a]
+
+ ALLWHS: trans o [WH(p,f) o [x,a],ALL g o (WH(p,f) o [x,a])]
+
+ => WH(p o [?1o?1,?2],[f o [?1o?1,?2],g o (f o [?1o?1,?2])]) o [[x,g o x],a]
+
+ *WH: trans o [*f o [x,t1 o (WH(p,g) o [y,a]),WH(p,g) o [y,a]] =>
+
+ WH(p o [?2o?1,?2],[f o [?1o?1,g o [?2o?1,?2]],g o [?2o?1,?2]]) o [[x,y],a]
+
+ *WHS: trans o [WH(p,g) o [y,a],*f o [x,t1 o (WH(p,g) o [y,a])] =>
+
+ WH(p o [?1o?1,?2],[g o [?1o?1,?2],f o [?2o?1,g o [?1o?1,?2]]) o [y,x],a]
+
+ +++++
```

première partie

Comme pour "\*", nous n'irons pas plus loin pour le moment.

### 3.6. Les axiomes sur "->\_;"

#### L'axiome C

C:  $p \rightarrow (q \rightarrow f; g); (q \rightarrow x; y) = q \rightarrow (p \rightarrow f; x); (p \rightarrow g; y)$

est permutatif. Pour introduire les axiomes sur "->\_;", il faut donc utiliser l'implémentation de la procédure de Knuth-Bendix "modulo" une théorie équationnelle. Pour cela, nous utiliserons la version de REVE 2.3 (qui est bientôt opérationnelle). En attendant, nous allons voir ce qu'on peut faire sans l'axiome C.

#### L'axiome COND1

COND1:  $(p \rightarrow f; g) o h = (p o h) \rightarrow (f o h); (g o h)$

Si on l'ordonne de droite à gauche, REVE diverge avec les superpositions de E1 (comme E2).

Si on l'ordonne de gauche à droite (avec la précedence + > ->\_;\_), COND1 s'ajoute à l'un quelconque de nos systèmes.

#### Les axiomes COND3 et COND4

COND3:  $p \rightarrow (p \rightarrow f; g); h = p \rightarrow f; h$

COND4:  $p \rightarrow h; (p \rightarrow f; g) = p \rightarrow h; g$

Ils s'orientent de gauche à droite (ce qui supprime un test inutile) et s'ajoutent, avec COND1, à tous nos systèmes.

première partie

L'axiome T

$$T: (p \rightarrow x; y) \rightarrow f; g = p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)$$

Cet axiome s'oriente soit de gauche à droite (avec un statut LR pour " $\rightarrow$ "), soit de droite à gauche (avec un statut RL pour " $\rightarrow$ ").

Dans les deux cas, il s'ajoute avec COND1 à tous nos systèmes. Mais avec l'axiome COND3 (ou COND4), REVE diverge.

Pour le choix de gauche à droite, T et COND3 se superposent à l'occurrence  $\epsilon$  donnant la paire critique

$$\begin{aligned} p \rightarrow (x \rightarrow (p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)); h); (y \rightarrow (p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)); h) \\ = p \rightarrow (x \rightarrow f; h); (y \rightarrow f; h) \end{aligned}$$

Cette paire critique s'oriente de gauche à droite en une nouvelle règle avec laquelle T se superpose de nouveau et cela continue indéfiniment.

Pour le choix de droite à gauche, la superposition de T et COND3 à l'occurrence  $\epsilon$  donne une paire critique

$$p \rightarrow f; (q \rightarrow f; g) = (p \rightarrow p; q) \rightarrow f; g$$

qui s'oriente de gauche à droite et se superpose sur T et COND3 donnant de nouvelles règles qui se superposent encore avec T et COND3.

L'introduction de l'axiome

PPX:  $p \rightarrow p; x = p \rightarrow T; x$  ne résout pas le problème.

L'axiome COND2

$$COND2: h \circ (p \rightarrow f; g) = p \rightarrow (h \circ f); (h \circ g)$$

première partie

Comme COND1, il s'ordonne de gauche à droite.

Il ne peut s'intégrer indépendamment de COND1. En effet, une paire critique avec E1:

$$(p \rightarrow (h \circ f); (h \circ g)) \circ z = h \circ ((p \rightarrow f; g) \circ z)$$

est détruite par COND1.

Or il se forme une paire critique avec COND1:

$$\begin{aligned} COND1COND2: (q \rightarrow (p \circ x); (p \circ y)) \rightarrow (q \rightarrow (f \circ x); (f \circ y)); (q \rightarrow (g \circ x); (g \circ y)) \\ = q \rightarrow ((p \circ x) \rightarrow (f \circ x); (g \circ x)); ((p \circ y) \rightarrow (f \circ y); (g \circ y)) \end{aligned}$$

qui est réduite en utilisant C (ainsi que T, COND3 et COND4)

Sans l'axiome C, nous ne pouvons pas intégrer l'axiome COND2.

Remarques

1) COND2 se superpose avec E2, ce qui donne une paire critique

$$p \rightarrow [x \circ f, y \circ f]; [x \circ g, y \circ g] = [p \rightarrow (x \circ f); (x \circ g), p \rightarrow (y \circ f); (y \circ g)]$$

qui est détruite par les axiomes COND5, COND3 et COND4.

2) COND2 se superpose avec E3 en donnant

$$SC: p \rightarrow x^-; x^- = x^- \quad SI \quad (DEF \circ p)$$

Les axiomes COND5

$$COND5_1: [p \rightarrow f; g, h] = p \rightarrow [f, h]; [g, h]$$

$$COND5_2: [h, p \rightarrow f; g] = p \rightarrow [h, f]; [h, g]$$

On étudie les deux orientations:

Première partie

- choix d'orientation de gauche à droite

Ces deux axiomes se superposent à l'occurrence  $\epsilon$  donnant une paire critique

$$\begin{aligned} \text{COND5COND5: } q \rightarrow (p \rightarrow [f, x]; [g, x]); (p \rightarrow [f, y]; [g, y]) \\ = p \rightarrow (q \rightarrow [f, x]; [f, y]); (q \rightarrow [g, x]; [g, y]) \end{aligned}$$

qui est permutative mais est détruite par C.

Même séparément, il faut COND2 pour détruire une paire critique obtenue avec P3

$$?1 \text{ o } (p \rightarrow [h, f]; [h, g]) = h \text{ SI } (\text{DEF o } (p \rightarrow [h, f]; [h, g]))$$

qui se réduit alors en une règle

$$S: p \rightarrow f; f \Rightarrow f \text{ SI } (\text{DEF o } (p \rightarrow f; f))$$

Remarquons que S détruit la paire critique SC issue de la superposition entre COND2 et E3.

- choix d'orientation de droite à gauche

Nous avons un bel exemple d'un cas où REVE échoue pour orienter une paire critique qui est détruite par une conséquence équationnelle qu'il découvre plus tard.

Les deux axiomes COND5 se superposent donnant une paire critique

$$\text{COND5COND5'} \quad [h, p \rightarrow f; f] = [p \rightarrow h; h, f]$$

qui n'est pas orientable. Mais si on force REVE à l'accepter avec, par

première partie

exemple une orientation de gauche à droite, la superposition avec P3 donne une nouvelle règle

$$S: p \rightarrow f; f \Rightarrow f \text{ SI } (\text{DEF o } (p \rightarrow f; f))$$

qui détruit la paire critique COND5COND5'. Si on commence par donner S, REVE ne nous contraint pas à orienter COND5COND5'.

Nous notons que les règles COND5 se superposent à tort à l'occurrence 1 sur la règle

$$T: (p \rightarrow x; y) \rightarrow f; g \Rightarrow p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)$$

car  $[h, x]$  et  $[h, y]$  ne peuvent être des prédicats. Si on rejette les paires critiques correspondantes, il n'y a plus de problème.

Les règles COND5 orientées par la précédence  $\_ \rightarrow \_ ; \_ > [ \_ ]$  s'ajoutent avec COND1, S et T (orienté dans un sens quelconque) à tous nos systèmes de réécriture.

Les règles COND5 se superposent sur COND3 (et COND4) donnant des paires critiques comme

$$p \rightarrow [p \rightarrow f; g, h]; z = p \rightarrow [f, h]; z$$

Ces paires critiques donnent des nouvelles règles en s'orientant de gauche à droite. Elles font diverger REVE par de nouvelles superpositions avec COND5.

Quelque soit le choix, on ne peut avoir les règles COND5 et COND3 (et COND4) ensemble.

première partie

### Les axiomes TRUE et FALSE

TRUE:  $T^- \rightarrow f; g = f$

FALSE:  $F^- \rightarrow f; g = g$

Ces axiomes s'orientent de gauche à droite et s'ajoutent avec COND1, COND3, COND4 et S à tous nos systèmes de réécriture.

### Avec la règle T

a) pour le choix

$T: p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g) \Rightarrow (p \rightarrow x; y) \rightarrow f; g$

ils se superposent à l'occurrence 1 (ou 2) donnant des paires critiques comme par exemple

$P: p \rightarrow f; (y \rightarrow f; g) = (p \rightarrow T^-; y) \rightarrow f; g$

On peut orienter ces paires de gauche à droite et, par superposition des règles TRUE et FALSE, on obtient de nouvelles règles

$p \rightarrow ((q \rightarrow T^-; F^-) \rightarrow x; y); z \Rightarrow p \rightarrow q; z$   
 $p \rightarrow ((q \rightarrow F^-; T^-) \rightarrow x; y); z \Rightarrow p \rightarrow z; q$

qui font diverger REVE. On pourrait détruire la première règle par l'axiome:

$TF: p \rightarrow T^-, F^- = p$

et la deuxième par l'axiome

$FT: (p \rightarrow F^-; T^-) \rightarrow x; y = p \rightarrow y; x$

Mais l'axiome FT se superpose sur T et fait diverger REVE.

L'orientation inverse des paires P fait aussi diverger REVE.

première partie

b) pour le choix inverse, il n'y a pas de problème. On obtient alors en intégrant aussi COND1, S et les axiomes COND5 (pour la précedence  $\_>\_;$ ;  $\_>\_$ ) des systèmes de réécriture S-CONDO qui peuvent inclure un quelconque des systèmes déjà trouvés.

```

+++++
+ +
+ S-CONDO +
+ ***** +
+ +
+ TOUT SYSTEME +
+ +
+ ***** +
+ +
+ COND1: (p->f;g) o h => (p o h)->(f o h);(g o h) +
+ +
+ COND5_1: p->[h,f];[h,g] => [h,p->f;g] +
+ +
+ COND5_2: p->[f,h];[g,h] => [p->f;g,h] +
+ +
+ S: p->f;f => f SI (DEF o (p->f;f)) +
+ +
+ T: (p->x;y)->f;g => p->(x->f;g);(y->f;g) +
+ +
+ TRUE: T^->x;y => x +
+ +
+ FALSE: F^->x;y => y +
+ +
+++++

```

### Les axiomes PPX et PXP

PPX:  $p \rightarrow p; x = p \rightarrow T^-; x$

PXP:  $p \rightarrow x; p = p \rightarrow x; F^-$

Ils s'orientent de gauche à droite (manuellement car il faut pouvoir mettre "T", "T" et "F" équivalentes et au bas de la précedence, ce que refuse la version de REVE que nous utilisons).

Avec COND1, COND3, COND4, TRUE et FALSE, il n'y a pas de problèmes. PPX se superpose sur la règle S donnant la nouvelle règle

$p \rightarrow T^-; p \Rightarrow p$

première partie

qui est réduite par PXP en

$$TF: p \rightarrow T^-; F^- \Rightarrow p$$

REVE se termine, on obtient des systèmes de réécriture S-COND1 avec un quelconque des systèmes de réécritures de paragraphes précédents.

```

+++++
+ +
+ S-COND1 +
+ ***** +
+ +
+ TOUT SYSTEME +
+ +
+ ***** +
+ COND1: (p->f;g) o h => (p o h)->(f o h);(g o h) +
+ +
+ COND3: p->(p->f;g);h => p->f;h +
+ +
+ COND4: p->h;(p->f;g) => p->h;g +
+ +
+ S: p->f;f => f SI (DEF o (p->f;f)) +
+ +
+ TRUE: T^->f;g => f +
+ +
+ FALSE: F^->f;g => g +
+ +
+ PPX: p->p;x => p->T^-;x +
+ +
+ PXP: p->x;p => p->x;F^- +
+ +
+ TF: p->T^-;F^- => p +
+ +
+++++

```

Entre la règle COND2 et PPX (ou PXP)

On obtient une paire critique

$$p \rightarrow (p \circ h); (x \circ h) = p \rightarrow (T^- \circ h); (x \circ h)$$

qui s'oriente de droite à gauche et provoque la divergence de REVE (avec par exemple les superposition de E1).

première partie

Entre les règles COND5 et PPX (ou PXP)

a) pour l'orientation correspondant à la précédence  $[_]>_->_-;$ :

PPX (ou PXP) se superpose sur

$$COND5_1: [h, p \rightarrow f; g] \Rightarrow p \rightarrow [h, f]; [h, g]$$

et donne la paire critique

$$p \rightarrow [h, T^-]; [h, x] = p \rightarrow [h, p]; [h, x]$$

Elle s'oriente de droite à gauche et se superpose aussi sur COND5. Cela continue indéfiniment.

b) pour l'orientation inverse:

REVE superpose PPX (ou PXP) sur les règles COND5. Mais la paire critique obtenue

$$[h, f] \rightarrow T^-; [h, g] = [h, [h, f] \rightarrow f; g]$$

est extensionnellement égale à "!" puisque  $[h, f]$  ne peut avoir pour résultat vrai ou faux. Si on rejette ces paires critiques, il n'y a plus de problème.

On peut ajouter PPX, PXP et les règles COND5 avec COND1, TRUE, FALSE, S et TF à l'un quelconque des systèmes de réécritures cités aux paragraphes précédents et REVE se termine. On obtient des systèmes de réécriture S-COND2.

première partie

```

+++++
+ S-COND2 +
+ ***** +
+ TOUT SYSTEME +
+ ***** +
+ COND1: (p->f;g) o h => (p o h)->f o h);(g o h) +
+ COND51: p->[h,f];[h,g] => [h,p->f;g] +
+ COND52: p->[f,h];[g,h] => [p->f;g,h] +
+ S: p->h;h => h SI (DEF o (p->h;h)) +
+ TRUE: T-->x;y => x +
+ FALSE: F-->x;y => y +
+ TF: p->T-;F- => p +
+ PPX: p->p;x => p->T-;x +
+ PPY: p->x;p => p->x;F- +
+++++

```

Entre la règle T et PPX (ou PXP)

a) pour le choix:

$$T: (p \rightarrow x; y) \rightarrow f; g \Rightarrow p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)$$

la superposition de PPX (ou PXP) à l'occurrence  $\epsilon$  donne une paire critique

$$p \rightarrow (x \rightarrow T^-; g); (y \rightarrow T^-; g) = p \rightarrow (x \rightarrow (p \rightarrow x; y); g); (y \rightarrow (p \rightarrow x; y); g)$$

qui serait détruite par C, COND3 et COND4.

b) pour le choix inverse,

PPX (ou PXP) et T se superposent à l'occurrence  $\epsilon$  en donnant une paire critique

première partie

$$(x \rightarrow f; g) \rightarrow T^-; (y \rightarrow f; g) = ((x \rightarrow f; g) \rightarrow x; y) \rightarrow f; g$$

qui s'oriente de gauche à droite, et fait diverger REVE.

#### 4. Utilisation des systèmes de réécriture trouvés

Nous pouvons utiliser les systèmes de réécriture trouvés au paragraphe 3 pour obtenir une forme normale des expressions fonctionnelles et en particulier pour celles qui sont données comme exemple au CHAPITRE I. Il suffit d'utiliser la commande REVE "normal-form" qui commande la réécriture d'un terme donné par un système résultat de la procédure de Knuth-Bendix.

Nous appliquons chaque système sur ces exemples et nous regardons l'amélioration apportée par la forme normale en utilisant des fonctions d'évaluation comme les fonctions N ou I utilisés au chapitre I.

Rappelons que I compte le nombre des séquences résultats des itérations pour une interprétation des expressions par une machine de type VON NEUMANN et que N y ajoute l'impact des primitives sur les séquences.

##### 4.1. Utilisation de S-BASE

###### Pour illustrer le rôle de S-BASE

Nous considérons, en limitant l'arité de "[\_]" à 2, l'exemple donné au chapitre 2, paragraphe 3.1, remarque 2.

L'expression

$$e = [f \circ ?1, g \circ ?2] \circ [x, y]$$

Première partie

se réécrit en

$e \downarrow_{S-BASE} = [f \circ x, g \circ y]$ , avec 5 étapes de réécriture.

On a évidemment  $N(e) = N(e \downarrow_{S-BASE})$  et  $I(e) = I(e \downarrow_{S-BASE})$ .

#### Rôle de $S-BASE$

La réécriture par une règle de  $S-BASE$  d'un terme  $t$  ne modifie pas  $N(t)$  et  $I(t)$  quand la règle est  $E1$ ,  $P1$ ,  $P2$ .

Il en est de même pour la réécriture par la règle  $E2$

$E2: [f, g] \circ h \Rightarrow [f \circ h, g \circ h]$

puisque le schéma d'exécution "partage" la fonction commune  $h$

Par ailleurs, les règles  $P3$  et la règle  $E3$

$E3: x^- \circ f \Rightarrow x^- \text{ SI } (DEF \circ f)$

$P3_1 ?1 \circ [f, g] \Rightarrow f \text{ SI } (DEF \circ [f, g])$

peuvent éventuellement faire décroître  $I$  ou  $N$ .

On peut noter que dans notre exemple l'application des règles  $P3$  n'a aucun effet sur  $I$  et  $N$ .

La réécriture par  $S-BASE$  diminue (ou laisse inchangé)  $I$  et  $N$   
 ~~~~~

#### 4.2. Utilisation de $S-PRIMITIVE$

##### Pour illustrer le rôle de $S-PRIMITIVE$

Nous prenons l'expression (2) de l'exemple 3 du chapitre I.

première partie

(2):  $/ \circ (\text{last} \circ (\text{trans} \circ [u, v]))$

where def  $u = *mult \circ [1^-, ALL \ 2^- \circ (\text{iota} \circ n)]$

and def  $v = *mult \circ [1^-, \text{iota} \circ n]$

(2) est réécrite en

$(2 \downarrow_{S-PRIMITIVE}): / \circ [\text{last} \circ u, \text{last} \circ v]$ , avec une étape de réécriture utilisant la règle  $LTL$ .

C'est l'expression que nous avons notée (3) au chapitre I.

Au chapitre I, on comptait  $N(2) = N(3) + 1$  et  $I(2) = I(3)$ .

#### Rôle de $S-PRIMITIVE$

$S-PRIMITIVE$  n'a pas d'influence sur  $I$ . Son rôle est plutôt de diminuer  $N$ .

Considérons les deux règles

$TLTRANS: t1 \circ (\text{trans} \circ [a, b]) \Rightarrow \text{trans} \circ [t1 \circ a, t1 \circ b]$

$APPDISTL: \text{distl} \circ [x, \text{append} \circ [a, b]]$

$\Rightarrow \text{append} \circ [\text{distl} \circ [x, a], \text{distl} \circ [x, b]]$

Il est clair que réécrire avec une de ces règles augmente le nombre des séquences intermédiaires apportées par les primitives sur les séquences.

Mais il est plus exact de tenir compte du coût "en place" des séquences intermédiaires.

On peut dire que la réécriture par  $TLTRANS$  améliore un programme: l'occurrence de "trans" a pour résultat une séquence plus courte dans le membre droit que dans le membre gauche.

première partie

On peut dire que la réécriture par APPDISTL n'a aucun effet sur le programme: "distl" donne dans le membre droit une séquence dont la longueur est exactement égale à la somme des longueurs des séquences résultats des deux occurrences de "distl" du membre gauche.

Pour en tenir compte, il suffit de différencier les valeurs accordées aux occurrences des primitives "tl", "trans" et "distl" selon la taille de la séquence résultat.

Comme cela, une réécriture par TLTRANS ou par APPDISTL n'a pas d'influence sur N et on peut dire que:

Les réécritures par S-BASE et S-PRIMITIVE diminuent (ou ne changent pas) N et ne changent pas I.

#### 4.3. Utilisation de S-ALL

Pour illustrer le rôle de S-ALL

##### EXEMPLE 1

Nous prenons le premier exemple cité au chapitre II paragraphe 3.2.

L'expression

(1): ALL mult o (trans o [all a o (iota o n), all b o (iota o n)])

se réécrit en

(1)<sub>S-ALL</sub>: ALL(mult o [a,b]) o (iota o n), avec 2 étapes de réécriture.

C'est l'expression que nous avons notée (2) au chapitre I. Au chapitre

première partie

I, on comptait:  $I(1) = 3$  et  $I(2) = 1$ ,  $N(1) = 4$  et  $N(2) = 1$ .

##### EXEMPLE 2

Il s'agit de l'expression (1) de l'exemple 3 du chapitre I:

(1): last o (ALL / o (trans o [u,v]))

la forme normale trouvée en appliquant S-ALL est l'expression

(1)<sub>S-ALL</sub>: / o [last o u, last o v]

obtenue avec 2 étapes de réécriture appliquant LALL' et LTRANS.

C'est l'expression que nous avons notée (3) au chapitre I et on comptait:

$I(1) = I(3) + 1$  et  $N(1) = N(3) + 2$ .

##### EXEMPLE 3

Nous considérons l'expression (2) de l'exemple 7 du chapitre I.

(2): last o (ALL / o u)

where def u = trans o [tl o (\*+ o [0<sup>-</sup>,x]),iota o 10<sup>-</sup>]

est réécrite en

(2)<sub>S-ALL</sub>: / o [last o (\*+ o [0<sup>-</sup>,x]), 10<sup>-</sup>], avec 3 étapes de réécriture.

On trouve l'expression que nous avons notée (3) pour cet exemple au chapitre I et l'on comptait:

$I(2) = 2$  et  $I(3) = 1$ ,  $N(2) = 4$  et  $N(3) = 1$ .

#### Rôle de S-ALL

première partie

A priori, S-ALL doit diminuer I (ou N) pour les séquences résultats des fonctions "ALL f". Mais tout dépend comment on évalue I (ou N).

● Le problème du "partage"

Introduisons le problème par un petit exemple:

$$t = [ALL\ f\ o\ ALL\ g, ALL\ g]$$

On compte  $I(t) = 2 + I(f) + I(g)$  (le schéma d'exécution partage "ALL g")

La forme normale est

$$t \downarrow_{S-ALL} = [ALL(f\ o\ g), ALL\ g]$$

pour laquelle on compte

a) si on ne partage plus g :  $I(t!) = 2 + I(f) + 2(I(g))$

b) si on continue de partager g :  $I(t!) = 2 + I(f) + I(g)$

REMARQUE

Soit  $t' = \text{trans } o\ t$

Sa forme normale est  $t' \downarrow_{S-ALL} = ALL[f\ o\ g, g]$

Et on compte  $I(t'!) = 1 + I(g) + I(f)$ .

Les deux occurrences de g se retrouvent arguments de "ALL" et partageables.

Le problème se pose avec les règles qui modifient la place (vis à vis d'une occurrence de "ALL") d'une variable entre le membre droit et le mem-

première partie

bre gauche.

EXEMPLES

1) HALL:  $hd\ o\ ALL\ f \Rightarrow f\ o\ hd$

On a par exemple

$[hd\ o\ ALL\ f, ALL\ f]$  évalué par  $1 + I(f)$

et  $[f\ o\ hd, ALL\ f]$  par

a)  $1 + 2I(f)$  (si f n'est plus partagé)

b)  $1 + I(f)$  (si f est toujours partagé)

2) FALL:  $\text{trans } o\ [ALL\ f, ALL\ g] \Rightarrow ALL[f, g]$

On a par exemple

$[\text{trans } o\ [ALL\ f, ALL\ g], ALL\ f]$  évalué par  $2 + I(f) + I(g)$

et  $[ALL[f, g], ALL\ f]$  par

a)  $2 + I(f) + 2I(g)$  (si f n'est plus partagé)

b)  $f : 2 + I(f) + I(g)$  (si f est toujours partagé)

Nous évaluons I (et N) avec des schémas d'exécution qui "partagent" sans tenir compte des occurrences de "ALL".

● TALL et ALLAPP

TALL:  $\text{trans } o\ [a, a] \Rightarrow ALL[id, id]\ o\ a$

Comme pour les règles TLTRANS et APPDISTL, il ne faut pas compter simplement le nombre d'occurrences de "ALL f": On doit compter la longueur de la séquence résultat.

ALLAPP:  $ALL\ f\ o\ (\text{append } o\ [a, b]) \Rightarrow \text{append } o\ [ALL\ f\ o\ a, ALL\ f\ o\ b]$

Il ne faut pas d'évaluer le premier membre par:  $2n_1 + 2n_2 + N(f)$  et le deuxième par:  $2n_1 + 2n_2 + 2N(f)$ . On doit en quelque sorte tenir compte de

première partie

l'application de  $f$  "n fois" sur une séquence de longueur  $n$  en comptant  $nN(f)$  et non  $N(f) + n$  pour une occurrence de "ALL  $f$ " ayant pour résultat une séquence de longueur  $n$ .

Si on calcule de cette manière, on peut dire que la réécriture par  $S\text{-ALL}$  diminue (ou ne change pas)  $N$ .

#### 4.4. Utilisation de $S\text{-*}$ (ou $S\text{-WH}$ ) et de $S\text{-**}$ (ou $S\text{-WHWH}$ )

##### Illustration du rôle de $S\text{-*}$

On considère l'expression (1) de l'exemple 1 du chapitre I

(1)  $\text{last } o (*\text{mult } o [1^-, \text{ALL } 2^-] o (\text{iota } o n))$

En utilisant  $S\text{-*}$  (ou  $S\text{-**}$ ), on trouve la forme normale

(2)  $\text{last } o (*(\text{mult } o [?1, 2^-]) o [1^-, \text{iota } o n])$ , avec 2 étapes de réécriture.

Au chapitre I, on comptait:

$$I(1) = 2 \text{ et } I(2) = 1, N(1) = 2 \text{ et } N(2) = 1.$$

A priori,  $S\text{-*}$  (ou  $S\text{-WH}$ ) doit diminuer  $N$  (pour le nombre des séquences intermédiaires résultat des itérations ALL  $f$  et  $*f$  (et  $\text{WH}(p, f)$ ). Nous retrouvons les problèmes d'évaluation du paragraphe précédent.

##### Illustration du rôle de $S\text{-**}$ (ou $S\text{-WHWH}$ )

On considère l'expression (2) de l'exemple 3 du chapitre I.

(2):  $/ o (\text{last } o (\text{trans } o [u, v])$

where def  $u = (*(\text{mult } o [?1, 2^-]) o [1^-, \text{iota } o n])$

and def  $v = *\text{mult } o [1^-, \text{iota } o n]$

En appliquant  $S\text{-**}$ , on trouve la forme normale ( $2^1_{S\text{-**}}$ ):

$/ o (\text{last } o (*[\text{mult } o [?1o?1, 2^-], \text{mult } o [?2o?1, ?2]] o [[1^-, 1^-], \text{iota } o n]))$   
avec 5 étapes de réécriture. C'est, modulo la commutativité de "mult", l'expression (5) que nous avons trouvée au chapitre I. Au chapitre I, on comptait:

$$I(2) = 2 \text{ et } I(5) = 1, N(2) = 3 \text{ et } N(5) = 1.$$

Ces systèmes permettent de faire les fusions d'itérations. Or, avec les fusions d'itérations " $*f$ " ou " $\text{WH}(p, f)$ ", le schéma du partage se complique.

Regardons par exemple l'axiome  $F*$

$F*: \text{trans } o [*f o [x, a], *g o [y, a]]$

$$\Rightarrow * [f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a]$$

Il faut continuer à partager  $f$  de l'itération du membre droit avec un  $f$  dans " $*f o [x, a]$ " pour évaluer:

$[\text{trans } o [*f o [x, a], *g o [y, a]], *f o [x, a]]$

et  $* [f o [?1o?1, ?2], g o [?2o?1, ?2]] o [[x, y], a], *f o [x, a]]$

par  $N([[x, y], a]) + nN(f) + nN(g) + 2n$

Si on évalue  $N$  à partir d'un schéma d'exécution qui réalise le partage convenablement, on peut dire que

Les systèmes de réécriture  $S*$  (et  $S\text{-WH}$ ),  $S\text{-**}$  (et  $S\text{-WHWH}$ ) diminuent (ou ne changent pas)  $N$ .

#### 4.5. Rôle des système S-CONDO, S-COND1 et S-COND2

##### Exemple

Si on considère l'expression

$$e = p1 \rightarrow (q1 \rightarrow [f, x_1]; [f, x_2]) ; (p2 \rightarrow (p2 \rightarrow [f, x_3]; [f, x_2]); [f, x_2])$$

elle est réécrite par S-CONDO ou S-COND2 en

$$e! = [f, p1 \rightarrow (q1 \rightarrow x_1; x_2) ; (p2 \rightarrow (p2 \rightarrow x_3 ; x_2); x_2)], \text{ avec 4 étapes de réécriture.}$$

Mais elle est réécrite par S-COND1 en

$$e! = p1 \rightarrow (q1 \rightarrow [f, x_1] ; [f, x_2]) ; (p2 \rightarrow [f, x_3] ; [f, x_2])$$

Dans les deux cas, si on partage  $f$ ,  $I$  ou  $N$  ne change pas. Mais la réécriture par S-COND1 a ôté un test  $p2$ .

##### Rôle de ces systèmes

Les règles sur " $\rightarrow$ ;" ne modifient pas  $I$  et  $N$  mais elles peuvent dans des cas limités (sur un niveau de " $\rightarrow$ ;" ) diminuer le nombre de tests par application des règles COND3, COND4 ou S.

#### 5. Conclusion

Arrivés à ce point, nous avons rempli une partie du programme que nous sommes proposés au chapitre I. Nous sommes capables de réduire le poids des séquences intermédiaires à l'intérieur des classes d'équivalences

première partie

définies par l'ensemble des axiomes que nous avons pu introduire dans chaque système.

Nous avons rassemblé dans l'annexe 4 l'effet de la réduction par le système S-\*\* sur les expressions données comme exemples au chapitre I dans l'annexe 4.

Nous ne pouvons pas encore minimiser les séquences intermédiaires pour tous les exemples du chapitre I. Récapitulons les problèmes qui ne sont pas encore résolus.

● Pour obtenir les systèmes de réécriture S-ALL, S-\*, S-\*\*, S-WH et S-WHH sont obtenus en orientant des règles "manuellement" et la preuve de terminaison n'a pas été faite par REVE. Elle reste à faire "manuellement". Nous nous posons ce problème dans la deuxième partie.

● Pour le système de réécriture S-PRIMITIVE, REVE se termine après rejet des équations non valides issues des superpositions de APPASS sur APPTRANS. Nous avons cependant observé que ces superpositions donnent une infinité d'autres équations valides et que l'on doit en tenir compte pour rendre le système convergent. Nous n'avons pas pu non plus inclure certains axiomes de la FP-théorie à cause de divergences de REVE. Toujours dans la deuxième partie, nous allons essayer de trouver une solution partielle pour ces divergences qui engendrent des infinités de règles en trouvant des codages pour certaines familles infinies de règles.

● Nous devons partir avec un ensemble fini d'axiomes, ce qui nous a obligé à limiter le nombre des sélecteurs et l'arité de " $[\ ]$ " à 2. Dans la deuxième partie, nous étendrons nos systèmes de réécriture à un nombre

première partie

illimité de règles tout en conservant leurs propriétés de convergence.

● Nous n'avons pas pu inclure certains axiomes de la FP-théorie parce qu'ils sont permutatifs. Il s'agit essentiellement de l'axiome C. Mais les axiomes sur les booléens rendraient plus utiles les règles PPX, PXP, TRUE et FALSE des systèmes S-COND. Les axiomes sur les primitives sur les entiers ou les réels ne sont pas inutiles si on veut utiliser les règles d'inférence du chapitre II. Nous verrons cela dans la deuxième partie.

● Il reste le problème de la réécriture par les règles conditionnelles (les règles dont la précondition nécessite une preuve) que nous envisagerons dans la deuxième partie.

## 6. Bibliographie

Dans le domaine des systèmes de réécriture, une présentation générale a été faite dans [Hue80b].

La propriété de confluence locale est rendue décidable par l'algorithme de Knuth-Bendix [Knu70]. Dans [Hue81] on peut trouver la preuve de correction de cet algorithme.

Prouver la propriété de terminaison finie d'un système de réécriture est plus difficile. Elle est indécidable. Nous avons cité la méthode de LANKFORD [Lan79] fondée sur le théorème de [Man70]. Les ordres de simplification sont plus faciles à implémenter. Nous avons cité l'ordre RPO ([Pla78], [Der82a]). L'ordre RDO ([Jou81b], [Jou82b], [Les81], [Les84a]) est implémenté dans le système REVE. Toute implémentation de la procédure de Knuth-Bendix exige un ordre pour orienter les équations.

REVE ([Les83],[For84]) est ici utilisé en tant qu'implémentation de la procédure de Knuth-Bendix. Nous avons vu que cette implémentation utilisait l'ordre RDO et sa propriété d'incrémentalité. La version utilisée est la version répertoriée 2.2 diffusée par le MIT et accessible sur le VAX du GRECO de Programmation.

## Plan de la deuxième partie

|                                                                                 |    |
|---------------------------------------------------------------------------------|----|
| TERMINAISON .....                                                               | 4  |
| 1. La méthode utilisée .....                                                    | 4  |
| 1.1. Les axiomes associativité et endomorphisme et la méthode de LANKFORD. .... | 5  |
| 1.2. Un exemple difficile .....                                                 | 7  |
| 2. Améliorations de la méthode de Lankford .....                                | 8  |
| 2.1. Méthode des "motifs" .....                                                 | 8  |
| 2.2. Distinction des opérations "surchargées" .....                             | 12 |
| 3. Les problèmes rencontrés au cours de la preuve de terminaison                |    |
| 3.1. Les règles E1, ALLALL et ALLALL' .....                                     | 14 |
| 3.2. La règle APPASS .....                                                      | 14 |
| 3.3. La règle TALL .....                                                        | 15 |
| 3.4. La règle *ALL .....                                                        | 15 |
| 3.5. L'axiome *LIST .....                                                       | 16 |
| 4. Conclusion                                                                   |    |
| DIVERGENCES .....                                                               | 18 |
| 1. Réécriture modulo un ensemble d'équations .....                              | 19 |
| 1.1. Définitions et résultats utiles .....                                      | 19 |
| 1.2. La procédure de complétion implémentée dans REVE 3 .....                   | 20 |
| 2. Schématisation d'une famille infinie de règles .....                         | 22 |
| 2.1. Exemple de schématisation d'une famille infinie de règles                  |    |
| 2.2. Un autre exemple .....                                                     | 27 |
| 3. Des méta-règles solution de nos problèmes de divergence ? .....              | 29 |

|                                                                                  |    |
|----------------------------------------------------------------------------------|----|
| 3.1. La confluence du système de réécriture sur les primitives                   |    |
| 3.2. Le système peut être complété avec l'axiome CONS .....                      | 31 |
| 3.3. Les axiomes ALL?1TRANS et ALL?2TRANS dans S-ALL .....                       | 34 |
| 4. Conclusion .....                                                              | 41 |
| 4.1. Un espoir d'améliorer nos résultats en ajoutant des méta-règles .....       | 41 |
| 4.2. Méthode de complétion dynamique [Glo80] .....                               | 42 |
| 4.3. Exploitation de cette idée                                                  |    |
| EXTENSION DES SYSTEMES DE REECRITURE AVEC LES INFINITES D'AXIOMES                |    |
| 1. Le problème .....                                                             | 45 |
| 1.1. Introduction .....                                                          | 45 |
| 1.2. Une extension d'un système de réécriture S .....                            | 47 |
| 2. Des systèmes de réécriture convergents .....                                  | 50 |
| 2.1. La propriété de confluence locale est-elle conservée? .....                 | 50 |
| 2.2. La propriété de terminaison finie est conservée. ....                       | 52 |
| 3. Schématisation des familles infinies de règles .....                          | 52 |
| 3.1. Schématisation des familles infinies engendrées par "[_...]" .....          | 60 |
| 3.2. Schématisation de familles infinies de règles incluant des sélecteurs ..... | 60 |
| 4. Conclusion                                                                    |    |
| FP-ALGEBRE TYPEE ET REECRITURE CONDITIONNELLE .....                              | 68 |
| 1. Règles à prémisses "DEF o t": un problème de contrôle de type.                |    |
| 1.1. Définition des types pour une procédure de "type checking"                  |    |
| 1.2. Systèmes de réécriture de règles à prémisses "t est bien formé" .....       | 74 |
| 2. Systèmes de réécriture conditionnels .....                                    | 75 |
| 3. Règles de la forme $m = n \mid -g \Rightarrow d$                              |    |

|                  |    |
|------------------|----|
| CONCLUSION ..... | 80 |
|------------------|----|

## TERMINAISON

Quand REVE oriente les équations en utilisant l'ordre RDO, il nous fournit la preuve de la terminaison des systèmes de réécriture obtenus. C'est le cas des systèmes de réécriture S-BASE, S-PRIMITIVE, et des systèmes S-COND que nous avons obtenus dans la première partie.

Pour les autres systèmes de réécriture, nous avons orienté les règles nous-mêmes en utilisant la possibilité d'orientation "manuelle" offerte par REVE. L'exemple type est celui de la règle ALLALL' que REVE oriente dans le sens

ALLALL':  $ALL\ f\ o\ (ALL\ g\ o\ h) \Rightarrow ALL(f\ o\ g)\ o\ h,$

orientation qui provoque une divergence immédiate de la procédure et qui ne correspond pas à notre volonté d'orienter les règles dans le sens qui permet de supprimer des séquences intermédiaires.

Nous allons dans ce chapitre exposer les problèmes posés par la preuve de terminaison des systèmes de réécriture que nous avons trouvés dans la première partie.

Nous allons faire cette preuve en utilisant des extensions de la méthode des polynômes, proposée par D. S. LANKFORD dans [Lan79], que nous avons citée au chapitre III paragraphe 1.2.

### 1. La méthode utilisée

est basée sur un théorème proposé par D. S. LANKFORD. Avant de rapeler

ce théorème, nous considérons l'exemple suivant:

### 1.1. Les axiomes associativité et endomorphisme et la méthode de LANKFORD.

On considère le système de réécriture localement confluent

$$(1) (x * y) * z \Rightarrow x * (y * z)$$

$$(2) f(x) * f(y) \Rightarrow f(x * y)$$

$$(3) f(x) * (f(y) * z) \Rightarrow f(x * y) * z.$$

L'ordre RDO ne permet pas de prouver la terminaison de ce système. Pour la règle (1) le statut de "+" doit être LR, ce qui donne pour la règle (3):

$$f(x * y) * z >_{RDO} f(x) * (f(y) * z).$$

Nous allons pourtant prouver la terminaison du système en utilisant la méthode de LANKFORD. Nous définissons pour cela une interprétation des termes et un ordre sur les termes.

#### a) Interprétation des termes

I est une interprétation des termes clos dans les entiers strictement positifs telle que I(f) conserve la même arité que le symbole fonctionnel f et que:

$$(C1) \ i < j \Rightarrow I(f)(x_1, \dots, i, \dots, x_n) < I(f)(x_1, \dots, j, \dots, x_n),$$

$$(C2) \ \text{si } f \text{ est d'arité } 1, \ I(f)(0) > 0.$$

Par abus de langage, on note I le morphisme associé à l'interprétation du symbole fonctionnel, ainsi

$$I(f(u_1, \dots, u_n)) = I(f)(I(u_1), \dots, I(u_n))$$

La condition (C1) peut aussi s'écrire

(C1)  $I(s) < I(t) \Rightarrow I(f(\dots, s, \dots)) < I(f(\dots, t, \dots))$ .

C'est la propriété de compatibilité de l'interprétation vis-à-vis des symboles de l'algèbre de [Man70]

#### b) Ordre sur les termes

On définit l'ordre  $t >_t u$

(  $t = f(t_1, \dots, t_n)$  et  $u = g(s_1, \dots, s_n)$  )

si et seulement si, pour toute substitution  $\gamma$

1)  $I(\gamma(t)) > I(\gamma(u))$

ou 2)  $I(\gamma(t)) = I(\gamma(u))$  et

a)  $f > g$

ou b)  $f = g$  et il existe un entier  $k$  inférieur ou égal à  $n$  tel que

$t_1 = s_1, t_2 = s_2, \dots, t_k = s_k$  et  $t_{k+1} >_t s_{k+1}$

#### c) Théorème [Len79]

Si les constantes sont interprétées par des entiers strictement supérieurs à 1, et si, pour toute règle  $g \Rightarrow d$  du système de réécriture, on a  $g > d$ , le système de réécriture est noethérien.

#### REMARQUE

Nous allons noter abusivement, pour un terme  $t$ ,  $I(\gamma(t))$  par  $I(t)$  et, pour une variable  $x$ ,  $I(\gamma(x))$  par  $I(x)$ . Cela n'a pas d'importance car la preuve est faite pour toute interprétation de  $x$  dans les entiers strictement positifs et elle reste donc la même pour toute substitution  $\gamma$ .

En utilisant ce théorème, nous allons prouver que le système "associativité et endomorphisme" est noethérien.

Si on définit l'interprétation

$I(x * y) = I(x) I(y)$

$I(f(x)) = n I(x) + 1$  (+ est ici l'addition dans les entiers et  $n$  est une constante entière strictement positive quelconque),

a) on trouve une interprétation égale pour les deux membres de (1) et on applique le cas 2.b de la définition de l'ordre des termes,

b) on trouve que les interprétations des premiers membres des règles (2) et (3) sont supérieures à celles de leurs seconds membres et on applique le cas 1 de la définition de l'ordre des termes.

On prouve donc que le système est noethérien.

#### 1.2. Un exemple difficile

On a le système de réécriture localement confluent  $R$

(1)  $(x * y) * z \Rightarrow x * (y * z)$

(2)  $(x + y) * z \Rightarrow (x * z) + (y * z)$

(3)  $(a * (x + y)) + z \Rightarrow x + (a * (y + z))$

("\*" n'est pas distributive à droite vis à vis de "+"). "a" est une constante.

L'ordre RDO ne permet pas de prouver la terminaison du système. Pour la règle (1), le statut de "\*" est LR et pour la règle (2), la précedence est

"\* > +". Dans ce cas, " $a * (x + y) + z$ " et " $a * (y + z)$ " deviennent incomparables.

Avec la méthode de LANKFORD, on prend l'interprétation

$$I(x * y) = I(x) I(y)$$

$$I(x + y) = I(x) + I(y)$$

qui donne une interprétation égale pour les deux membres de chaque équation (1) et (2). On applique le cas 2.b de l'ordre des termes pour (1) et le cas 2.a de l'ordre des termes avec " $* > +$ " pour (2).

Mais on ne peut rien conclure pour l'équation (3) car l'interprétation de la constante  $a$  doit être supérieure à 1 ce qui ne permet pas d'avoir l'interprétation du membre gauche supérieure ou égale à celle du membre droit pour toutes les substitutions des variables.

## 2. Améliorations de la méthode de Lankford

### 2.1. Méthode des "motifs"

Considérons l'exemple précédent

- (1)  $(x * y) * z \Rightarrow x * (y * z)$
- (2)  $(x + y) * z \Rightarrow (x * z) + (y * z)$
- (3)  $(a * (x + y)) + z \Rightarrow x + (a * (y + z))$ ,

nous allons transformer les termes. Nous considérons le "motif"

$$m = a * (x + y)$$

Deuxième partie

que nous remplaçons par  $m' = A(x, y)$ .

Nous pouvons définir un morphisme d'algèbres, noté  $\Phi$  (de  $T(S, V)$  dans  $T(SU\{A\}, V)$ ).  $\Phi$  est tel que  $\Phi(m) = m'$ . Par application de  $\Phi$  à leurs membres gauches et droits, les règles (1) et (2) sont inchangées mais la règle (3) devient (3)'

$$(3)': A(x, y) + z \Rightarrow x + A(y, z)$$

Nous voulons que les chaînes de réécritures dans le nouveau système  $R' = \{(1), (2), (3)'\}$  restent duales de celles de l'ancien système  $R$ , c'est-à-dire que, si

$$t_0 \Rightarrow t_1 \Rightarrow t_2 \Rightarrow \dots \Rightarrow t_n$$

est une chaîne de réécriture de  $R$ , on a aussi

$$t_0' \Rightarrow t_1' \Rightarrow t_2' \Rightarrow \dots \Rightarrow t_n'$$

pour  $t_i' = \Phi(t_i)$  pour  $i=0, 1, 2, \dots, n$  dans  $R'$ .

Pour cela, il faut avoir la propriété de commutation entre  $R$  et  $R'$ , propriété qui a été mise en évidence dans le cadre plus général de la terminaison de règles modulo des équations par [Jou84b].

#### Propriété de commutation

Soit deux systèmes de réécriture  $R_1$  de l'algèbre  $A_1 = T(S, V)$  et  $R_2$  de l'algèbre  $A_2 = T(\{A\} \cup S, V)$  et un morphisme  $\Phi$  de  $A_1$  dans  $A_2$ , si pour

$$t_1 \Rightarrow t_2 \text{ dans } R_1, \text{ on a aussi}$$

$$t_1' \Rightarrow^* t_2' \text{ pour } t_1' = \Phi(t_1) \text{ et } t_2' = \Phi(t_2), \text{ dans } R_2,$$

Deuxième partie

nous disons que les systèmes  $R_1$  et  $R_2$  commutent pour  $\Phi$ .

Remarque:

Il se peut que, dans un terme, l'application de l'associativité (1) ou de la distributivité (2) fasse apparaître le motif  $m$ . Aussi, nous allons ajouter à  $R'$  la règle  $rm$ :

$(rm) : m \Rightarrow m'$

#### Théorème

Si la propriété de commutation est satisfaite entre  $\{rm\} \cup R'$ , et si  $\{rm\} \cup R'$  termine, alors  $R$  termine.

Preuve: S'il existe une chaîne infinie de réécriture dans  $R$ , elle existe aussi dans  $R' \cup \{rm\}$ .

Pour que la propriété de commutation soit satisfaite, il ne faut pas que l'application de  $\Phi$  modifie la réécriture des termes. Cela n'est pas le cas pour notre exemple car  $m = a * (x + y)$  se superpose sur la règle (1) pour donner le terme

$t = (a * (x + y)) * z$  qui peut se réécrire en

$t' = a * ((x * z) + (y * z))$  en utilisant les règles (1) et (2)

alors que  $\Phi(t) = A(x, y) * z$  ne le peut pas.

Si l'on ajoute au système  $R$  la règle  $(rm)$ , la règle

$(r0) : A(x, y) * z \Rightarrow A(x * z, y * z)$ .

Deuxième partie

est obtenue par complétion.

On peut remplacer, pour un terme  $\Phi(t)$ , l'application de  $r0$  par l'application des deux règles (1) et (2) pour le terme  $t$ . Nous dirons que (2) est la règle duale de  $r0$ .

Nous allons montrer que le système de réécriture  $R$  commute avec le système  $R' \cup \{rm, r0\}$ .

Nous avons alors un nouveau moyen pour prouver la terminaison, en prouvant la terminaison du système  $\{(1), (2), (3'), r0, rm\}$ .

#### Principe d'utilisation des "motifs"

• Soit  $R$  le système de réécriture  $\{g \Rightarrow d\}$  de  $T(S, V)$  dont on veut prouver la terminaison,

• Soit un terme "motif"  $m$  et  $\Phi$  un morphisme des FP-termes tel que  $\Phi(m) = m'$  et appelons  $rm$  une règle  $m \Rightarrow m'$

• On étend le morphisme  $\Phi$  aux règles  $r: g \Rightarrow d$  en définissant

$\Phi(r) : \Phi(g) \Rightarrow \Phi(d)$ .

• Pour une superposition  $\sigma(g/u) = \sigma(m)$  du motif sur un membre gauche d'une règle  $ra: g \Rightarrow d$  de  $R$ , on définit une règle

$r0 : \sigma(g[u \leftarrow m']) \Rightarrow \Phi(\sigma(d)) \downarrow_R$

telle qu'il existe une règle  $d0$  "duale" de  $r0$  pour normaliser " $\Phi$ "

$\Phi(d) \Rightarrow d0 \Phi(d) \downarrow_R$ .

Deuxième partie

Si le système de réécriture  $\{r_0, r_m\} \cup R'$  (pour  $R' = \{\phi(g \Rightarrow d)\}$ ) est noethérien alors, le système de réécriture  $R$  est noethérien.

Preuve:  $R$  et  $R' \cup \{r_m, r_0\}$  commutent pour  $\phi$ .

On a les deux cas:

-  $t_k \Rightarrow^r t_{k+1}$  et  $\phi(r)$  réécrit  $t_k'$  en  $t$

et  $t \Rightarrow^{r_m} t_{k+1}'$  (car  $V(m) = V(m')$ )

- pour  $r = r_a$  correspondant à la superposition du motif, il existe  $t$  tel que  $t_{k+1} \Rightarrow^{d_0} t$  si  $t_k' \Rightarrow^{r_0} t_{k+1}'$ .

donc, dans les deux cas, la propriété de commutation est satisfaite.

Exemple:

Nous pouvons montrer maintenant la terminaison du système de règles (1), (2), (3'), (r0) et  $r_m : a * (x + y) \Rightarrow A(x, y)$  en utilisant l'interprétation

$$I(x + y) = I(x) + I(y), I(x * y) = I(x) I(y), I(A(x, y)) = I(x) + I(y),$$

avec la méthode de LANKFORD, il suffit de prendre " $*$  >  $+$ " et " $*$  >  $A$ ".

## 2.2. Distinction des opérations "surchargées"

Soit la règle

$$(6) \quad c * f(x) \Rightarrow f(x) * c \quad (c \text{ est une constante})$$

Si  $f$  correspond à deux symboles  $f_1, f_2$  et si à la règle (6) correspond en fait la règle

Deuxième partie

$$(6)' \quad c * f_2(x) \Rightarrow f_1(x) * c$$

il est facile de prouver la terminaison de (6)' en prenant par exemple

$$I(f_2(x)) = 2x + 1 \text{ et } I(f_1(x)) = x + 1$$

et on prouve la terminaison de (6) pour les termes "bien formés" c'est à dire ceux pour lesquels il est possible de décider si  $f$  est l'opération correspondant à  $f_1$  ou celle qui correspond à  $f_2$ .

## Le principe est le suivant

Soit un symbole  $s \in S$  de l'algèbre  $T(S, V)$  correspondant à une opération "surchargée",

soit  $\Delta$  le morphisme de  $T(S', V)$  dans  $T(S, V)$  défini par  $\Delta(s_1) = \Delta(s_2) = \dots = s$ ,

soit  $R = \{g \Rightarrow d\}$ , un système de réécriture de  $T(S, V)$ ,

on considère le système de réécriture  $R' = \Delta^{-1}(R)$  de  $T(S', V)$  ( $R'$  peut avoir une infinité de règles si  $s$  correspond à une infinité de symboles).

La réécriture dans  $R'$  est duale de la réécriture dans  $R$  restreinte aux termes "bien formés" de  $T(S, V)$ . Aussi il suffit de prouver que  $R'$  est noethérien pour assurer la terminaison de la réécriture restreinte aux termes "bien formés" dans  $R$ .

## 3. Les problèmes rencontrés au cours de la preuve de terminaison

Le détail de la preuve de terminaison de nos systèmes de réécritures est

Deuxième partie

très technique. Cette preuve est faite dans l'annexe 5. Nous nous contentons ici de signaler les points les plus délicats.

### 3.1. Les règles E1, ALLALL et ALLALL'

E1:  $(f \circ g) \circ h \Rightarrow f \circ (g \circ h)$

ALLALL:  $ALL f \circ ALL g \Rightarrow ALL(f \circ g)$

ALLALL':  $ALL f \circ (ALL g \circ h) \Rightarrow ALL(f \circ g) \circ h$

On est dans le cas de l'exemple des axiomes d'associativité et d'endomorphisme que nous avons signalé au paragraphe précédent.

### 3.2. La règle APPASS

APPASS:  $append \circ [append \circ [x,y],z] \Rightarrow append \circ [x,append \circ [y,z]]$

Comme on doit avoir  $I(append) > 1$ , on ne peut pas utiliser directement la méthode de LANKFORD. Nous considérons le "motif"  $append \circ [x,y]$  et un morphisme  $\Phi$  tel que

$$\Phi(append \circ [x,y]) = A(x,y)$$

rm:  $append \circ [x,y] \Rightarrow A(x,y)$ .

La superposition sur l'associativité E1 donne la règle

r0:  $A(x,y) \circ z \Rightarrow A(x \circ z, y \circ z)$  avec comme règle duale la distributivité E2.

$\Phi(APPASS): A(A(x,y),z) \Rightarrow A(x,A(y,z))$

La terminaison est alors prouvable par la méthode de LANKFORD en prenant

Deuxième partie

$$I(A(x,y)) = I(x) + I(y) \text{ et } "o > A".$$

### 3.3. La règle TALL

TALL:  $t1 \circ ALL f \Rightarrow ALL f \circ t1$  SI  $(DEF \circ t1)$

Nous distinguons une infinité de symboles ALL: ALL1, ALL2, ... selon la longueur de la séquence à laquelle ALL f s'applique.

$$\Delta^{-1}(TALL) = \{t1 \circ ALL_n f \Rightarrow ALL_{n-1} f \circ t1\}$$

et interprétons les symboles  $ALL_n$  par  $I(ALL_n) = n I(f) + 1$ .

### 3.4. La règle \*ALL

\*ALL:  $*f \circ [x, ALL g \circ h] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, h]$

La méthode de LANKFORD n'est pas utilisable. On considère le "motif"

$m = *k \circ [y, 1]$  et le morphisme  $\Phi$  tel que

$$\Phi(m) = STAR(k, y, 1).$$

La superposition sur l'associativité E1 donne

STAR:  $STAR(k, y, 1) \circ z \Rightarrow STAR(k, y \circ z, 1 \circ z)$  avec comme règle duale E2.

\*ALL:  $STAR(f, x, ALL g \circ h) \Rightarrow STAR(f \circ [?1, g \circ ?2], x, h)$

La terminaison est alors prouvable par la méthode de LANKFORD avec

$$I(STAR(f, x, a)) = I(x) + I(f) I(a) \text{ et } "o > STAR".$$

Deuxième partie

### 3.5. L'axiome \*LIST

\*LIST:  $*f \circ [x, L(y)] \Rightarrow \text{append } o [L(x), f \circ [x, y]]$

devient par application de  $\Phi$

$\Phi(*LIST): \text{STAR}(f, x, L(y)) \Rightarrow A(L(x), f \circ [x, y])$

Si on appelle l et r les membres gauche et droit, on a, compte tenu des interprétations précédentes

$$I(l) = I(x) + I(f) I(L(y)) \text{ et}$$

$$I(r) = I(L(x)) + I(f) (I(x) + I(y))$$

Nous n'avons pas trouvé le moyen de prouver la terminaison d'un système qui contient à la fois \*ALL et \*LIST.

### 4. Conclusion

L'annexe 5 donne le détail de la preuve que, les systèmes localement confluents S-BASE, S-PRIMITIVE, S-\*\*, S-WHH et S-COND que nous avons trouvés dans la première partie sont noéthériens. La terminaison des systèmes S-\* et S-WH reste une conjecture car nous n'avons pas pu encore trouver une interprétation satisfaisante des termes " $*f \circ [x, a]$ " (et " $WH(p, f) \circ [x, a]$ ") qui nous permette de faire la preuve avec des systèmes qui contiennent à la fois les axiomes \*LIST et \*ALL.

L'essentiel, pour l'immédiat, est que nous ayons pu prouver la terminaison des systèmes S-\*\* et S-WHH qui contiennent les règles de fusion

et de composition des itérations et qui sont ceux qui servent à la transformations de nos exemples (voir l'annexe 4).

Cependant un travail doit être fait pour simplifier les preuves. Faire manuellement la preuve de terminaison d'un système de réécriture résultat de la procédure de Knuth-Bendix n'est pas une bonne solution. La technicité des preuves n'emporte pas facilement l'adhésion.

Une étude générale des problèmes de terminaison rencontrés ici a démarré depuis peu, dans le cadre de la terminaison de règles modulo des équations comme par exemple l'associativité et la commutativité [Bac85]. Ces travaux ont été repris et étendus [Gna85] avec l'idée que ces méthodes s'appliquent également au cas de la terminaison traditionnelle. On peut espérer que l'ensemble des problèmes exposés ici pourra être très prochainement résolu dans un formalisme unique intégrant aussi bien la méthode des motifs que la preuve de terminaison des axiomes d'associativité, commutativité et endomorphisme.

Signalons aussi que l'implémentation des ordres définis avec des interprétations des termes par les polynômes fait l'objet d'une thèse à paraître prochainement [Che85].

## DIVERGENCES

Nous n'avons pas pu trouver un système de réécriture convergent qui soit équivalent à l'ensemble des axiomes que nous avons prévus pour la FP-algèbre. Au cours de notre expérimentation avec REVE, il ne nous a pas été possible d'éviter de faire diverger REVE en voulant inclure certains d'entre eux. Pourtant l'on sait, théoriquement [Hue81], que, avec l'infinité des règles engendrées par la procédure de Knuth-Bendix, le système de réécriture est convergent.

L'idée d'introduire un schéma de règle qui est capable de décrire (c'est à dire de schématiser) une famille infinie de règles engendrées par la procédure de Knuth-Bendix a été introduite par [Kir82]. Il faut savoir schématiser (c'est à dire associer) à une famille infinie de règles un schéma de règle, qui s'appelle une méta-règle. Il faut aussi, avec la méta-règle, savoir, comme si on avait toutes les règles de la famille, réécrire et tester si le système de réécriture est complet.

Quand la procédure de Knuth-Bendix diverge, on peut vouloir inférer automatiquement la méta-règle. Les outils développés pour la synthèse des programmes à partir d'exemples [Jou81a] pourraient être utilisés pour cela. Cependant, ces outils sont loin d'être implémentés en liaison avec une procédure de Knuth-Bendix.

Dans ce chapitre, nous inférons "manuellement" les méta-règles qui permettent de résoudre quelques unes des divergences que nous avons rencontrées lors de notre expérimentation. Nous utilisons le formalisme proposé par H. KIRCHNER [Kir85a].

La validité d'une méta-règle, c'est à dire l'équivalence entre la méta-règle et la famille infinie engendrée par la procédure de Knuth-Bendix peut être prouvée en utilisant la procédure de méta-complétion décrite par H. KIRCHNER. Cette procédure, teste que le méta-système de réécriture est convergent pour une relation de méta-réécriture, ce qui prouve la correction de la schématisation. L'implémentation de cette procédure est nécessaire pour valider les méta-règles que nous découvrons dans ce chapitre.

La méta-réécriture est liée à la réécriture modulo un ensemble d'équations  $E$  ([Hue80a],[Hu180],[Der83],[Jou84a],[Jou84b]).

Nous allons commencer par rappeler brièvement ce qu'est la réécriture modulo  $E$  puis nous préciserons ce qu'est une méta-règle avec le formalisme de H. KIRCHNER. Nous donnons ensuite des exemples de divergence que nous schématisons manuellement et qui pourront sans doute être résolus par utilisation de méta-règles.

### 1. Réécriture modulo un ensemble d'équations

#### 1.1. Définitions et résultats utiles

Nous rappelons succinctement quelques définitions et quelques résultats.

La réécriture modulo  $E$  qui est notée " $\Rightarrow_R/E$ " est la réécriture modulo la classe d'équivalence de  $E$ -égalité:

$$\Rightarrow_R/E =_{E'} \Rightarrow_R =_E$$

On sait si un terme est réductible modulo E quand les classes d'équivalences modulo E sont finies.

Un algorithme de complétion modulo E a été défini par G. HUET [Hue80]. Il n'exige pas d'algorithme de E-unification mais le système de réécriture doit être linéaire à gauche.

Pour parler des méta-règles, nous avons besoin de la définition de la R,E réécriture.

Un terme t est R,E réductible s'il existe une occurrence u, une règle  $g \Rightarrow d$  dans R et une substitution  $\sigma$  telles que

$$t/u =_E \sigma(g).$$

Si  $t' = t[u \leftarrow \sigma(d)]$ , on note  $t \Rightarrow_{R,E} t'$ .

Un algorithme de E-filtrage est nécessaire pour la R,E réécriture.

On a  $\Rightarrow_{R,E} \subseteq \Rightarrow_{R/E}$

En utilisant la R,E réécriture, on peut définir une procédure de complétion qui permet d'obtenir des systèmes de réécriture E-convergeants. Mais cette procédure exige un algorithme de E-unification [Pet81].

### 1.2. La procédure de complétion implémentée dans REVE 3

Une autre procédure est définie par [Jou84a], Les auteurs utilisent la propriété de E-cohérence d'une relation de réécriture R':

Appelons  $t \downarrow_R$ , la forme normale d'un terme t pour la relation R', R' est E-

Deuxième partie

cohérente si et seulement si, pour tous termes  $t_1, t_2$  et  $t_3$  tels que

$$t_1 =_E t_2 \quad \text{et} \quad t_1 \Rightarrow_{R'} t_2 \Rightarrow^*_{E,R} t_3,$$

il existe  $t_4$  tel que  $t_2 \Rightarrow_{R'} t_4$  et  $t_3 \downarrow_{R'} =_E t_4 \downarrow_{R'}$ .

La E-cohérence et la E-confluence de R,E implique la R/E confluence, ce qui permet de définir une procédure de complétion qui unifie les deux précédentes.

Cette procédure est implémentée dans la version REVE 3. Elle exige un algorithme de E-unification.

Signalons qu'il faut utiliser cette version REVE 3 pour introduire les axiomes permutatifs de la FP-algèbre. En particulier, nous ne pouvons inclure les axiomes sur " $\rightarrow$ ;" sans l'axiome C qui est permutatif:

$$C: p \rightarrow (q \rightarrow f; g); (q \rightarrow x; y) = q \rightarrow (p \rightarrow f; x); (p \rightarrow g; y).$$

### REMARQUES

Il sera nécessaire de trouver une procédure de C-unification,

Dans le contexte de REVE 3, C. KIRCHNER propose un cadre pour trouver des algorithmes d'unification ([Jou82a], [Jou83], [Fag83], [Kir84], [Kir85b]). Il projette d'essayer de compléter les axiomes de la conditionnelle ([Blo81], [Gue77]) pour pouvoir décider l'égalité des conditionnelles.

On peut sans doute simplifier le problème en partageant l'axiome C en deux

$$C1: \text{DEF } o \quad q \rightarrow (p \rightarrow f; h); (p \rightarrow g; h) :: q \rightarrow (p \rightarrow f; h); (p \rightarrow g; h) = p \rightarrow (q \rightarrow f; g); h$$

$$C2: \text{DEF } o \quad q \rightarrow (p \rightarrow h; f); (p \rightarrow h; g) :: q \rightarrow (p \rightarrow h; f); (p \rightarrow h; g) = p \rightarrow h; (q \rightarrow f; g)$$

C1 et C2 sont des conséquences équationnelles de C et S.

Deuxième partie

C est conséquence équationnelle de C1,C2,COND3 et COND4.

Notons que si on utilise REVE 2 avec les axiomes C1,C2,COND3 et COND4, on retrouve C comme paire critique, ce qui ne nous avance guère.

Mais il est sans doute plus simple de réécrire modulo C1 et C2 plutôt que de réécrire modulo C. Signalons d'ailleurs que, dans le but de rechercher des algorithmes de filtrage pour les axiomes de " $\_ \rightarrow \_ ; \_$ ", J. S. GIVLER et R. B. KIEBURZ [Giv84] utilisent C1 et C2 et non C.

## 2. Schématisation d'une famille infinie de règles

Nous allons introduire informellement le processus de schématisation proposé par H. KIRCHNER [Kir85a]. Nous utilisons un exemple.

### 2.1. Exemple de schématisation d'une famille infinie de règles

Nous considérons un exemple simple de divergence de la procédure de Knuth-Bendix. Les axiomes sont:

- Les axiomes de distributivité et d'associativité entre deux opérations "+" et "\*";

$$\begin{aligned} ((x+y)+z) &= (x+(y+z)) \\ (x+y)*h &= (x*h)+(y*h) \text{ et} \end{aligned}$$

- un axiome de "pseudo-distributivité entre "+" et une opération "."

$$(x+a).h \Rightarrow x*h + a$$

où "a" est une constante.

Deuxième partie

Ces axiomes, orientés de gauche à droite, engendrent la famille infinie de règles

$$\begin{aligned} (x_1+a).h &\Rightarrow x_1*h + a, \\ (x_1+(x_2+a)).h &\Rightarrow (x_1*h + (x_2*h + a)), \\ (x_1+(x_2+(x_3+a))).h &\Rightarrow (x_1*h + (x_2*h + (x_3*h + a))), \dots \end{aligned}$$

A l'aide de cet exemple, nous allons suivre pas à pas le processus de schématisation d'une méta-règle, ce qui nous permettra d'introduire les définitions et résultats utiles de façon intuitive.

#### a) Schématisation des membres gauche par un "méta-terme"

Nous considérons la suite des termes  $\langle g_k \rangle$  que sont les membres gauches des règles de la famille infinie que nous avons l'intention de schématiser.

La suite  $g_k$  est schématisable si l'on peut trouver un "squelette" commun à tous les termes de la suite qui est ici

$$(x+a).h$$

ainsi que des structures, c'est à dire des ensembles de termes, qui sont ici

$$x_2, x_2+x_3, x_2+(x_3+x_4), \dots$$

Les structures vont se substituer à la variable x pour former ce qu'on appellera la "branche de x" notée " $B_x$ ".

Tout cela n'est utilisable que si chaque branche engendre un ensemble de termes que l'on appelle un domaine et que l'on sait reconnaître.

® Pour une suite  $\langle g_k \rangle$ , on définit les structures S qui vont se substituer (au renommage des variables près) à la variable x pour former la

Deuxième partie

"branche de x"

$$B_x = \{y_k(x)\}$$

● Soit  $R_0$  le système de réécriture convergent formé par l'axiome d'associativité,

● le domaine engendré par la branche " $B_x$ " est l'ensemble des termes  $t$  réductibles par  $R_0$  en une substitution d'un des termes de l'ensemble " $B_x$ ".

Pour l'exemple, c'est particulièrement simple car les structures comprennent une variable, aussi, le domaine engendré par les structures est l'ensemble des termes. Il est donc décidable.

● Pour tout élément de la suite  $(g_k)$ , il existe une substitution  $\gamma_k$  par un terme (c'est à dire un élément du domaine) telle que

$$\gamma_k((x + a).h) \Rightarrow^{*R_0} g_k$$

Le "squelette  $(x+a).h$ " généralise la suite  $g_k$ .

La suite  $g_k$  est maintenant schématisable.

~~~~~

Le squelette de la suite  $(g_k)$  est le terme " $(x + a).h$ " dans lequel nous allons remplacer la variable  $x$ , telle que  $\{y_k(x)\} = S$ , par une méta-variable  $X$ .

Les méta-variables sont notées par des majuscules.

Le squelette est un méta-terme. Il s'écrit

$$(X + a).h.$$

Deuxième partie

Le domaine de la méta-variable  $X$  est ici l'ensemble des termes.

$X$  est instanciable uniquement par des éléments de son domaine. Une instantiation de  $X$  par une structure est une instantiation principale.

Le membre gauche d'une méta-règle est le méta-terme squelette.

Pour notre exemple, la méta-règle s'écrit donc

$$(X+a).h \Rightarrow X*h + a.$$

La réécriture avec la méta-règle ou "méta-réécriture" doit tenir compte de l'instanciation particulière des méta-variables dans leur domaine.

#### b) méta-réécriture

Nous voulons maintenant qu'une méta-règle schématise la famille infinie de règles et, pour cela nous avons besoin de nouvelles relations de réécriture.

● Soit  $MR$  la méta-règle  $L \Rightarrow R$ , un terme  $t$  se réécrit en  $t'$  selon la réécriture contrainte modulo un système de réécriture convergent  $R_0$ , ce qui se note

$$t \Rightarrow t',$$

si et seulement si il existe une occurrence  $u$  de  $t$ , une substitution  $\gamma$  qui instancie les méta-variables dans leur domaine et telle que  $\gamma$  soit un  $R_0$ -filtre de  $L$  vers  $t/u$  avec

$$t' = t[u \leftarrow \gamma(R)]$$

$$\text{On a } \Rightarrow \subseteq \Rightarrow^{MR, R_0}$$

Deuxième partie

Pour notre exemple, le domaine étant l'ensemble des termes, pour  $RO$ , la réécriture contrainte modulo  $RO$  coïncide avec la  $MR, RO$  réécriture.

C'est encore insuffisant. On a

$$(x_1 + (x_2 + (x_3 + a))).h \Rightarrow (x_1 + (x_2 + x_3)) * h + a$$

et nous voudrions obtenir

$$(x_1 * h + (x_2 * h + (x_3 * h + a))).$$

Pour cela, il faut réécrire avec l'axiome de distributivité.

Nous incluons dans  $RO$ , l'axiome de distributivité.  $RO$  est convergent.

● On appelle relation de méta-réécriture notée " $\Rightarrow$ " la relation

$$\Rightarrow = \Rightarrow_{RO}$$

On a  $\Rightarrow \subseteq \Rightarrow_{MR/RO}$

On a alors:

$$(x_1 + (x_2 + (x_3 + a))).h \Rightarrow (x_1 * h + (x_2 * h + (x_3 * h + a)))$$

Appelons  $(d_k)$  la suite des membres gauches de la famille infinie de règles que nous considérons:

La famille infinie des règles " $g_k \Rightarrow d_k$ " est schématisable par la

~~~~~  
 méta-règle  $(X+a).h \Rightarrow X*h + a \quad \text{modulo } RO$   
 ~~~~~

Le domaine de la méta-variable  $X$  est l'ensemble des termes.

Deuxième partie

La suite  $(g_k)$  est schématisable par le squelette  $L$  qui est le membre droit de la méta-règle.

Pour tout  $k$ ,  $g_k \Rightarrow^{+} d_k$

~~~~~

Au sujet de la méta-complétion, nous renvoyons le lecteur aux résultats d'H. Kirchner [Kir85a]. Signalons que c'est la procédure de méta-complétion qui permet de vérifier que la schématisation de l'ensemble infini de règles par la méta-règle est correcte, c'est à dire que la méta-règle peut jouer le même rôle dans le système de réécriture que l'ensemble infini de règles.

Avec une procédure de méta-complétion, nous pouvons savoir si nos  
 ~~~~~  
 méta-règles sont correctes.  
 ~~~~~

Indiquons seulement les contraintes exigées par cette procédure:

- la réécriture contrainte modulo  $RO$  exige un algorithme de  $RO$  filtrage et, dans ce cas,  $\Rightarrow$  est décidable.

- la méta-unification exige un algorithme de  $RO$ -unification.

## 2.2. Un autre exemple

Considérons l'ensemble des règles  $S-ALL$  auquel on ajoute la règle

$ALL?1TRANS: ALL ?1 o (trans o [a,b]) \Rightarrow a \text{ SI } (DEF o (trans o [a,b])).$

Par superposition avec la règle  $ALLALL'$

$ALLALL': ALL f o (ALL g o h) \Rightarrow ALL(f o g) o h,$

Deuxième partie

on obtient la règle:

$$\text{ALL}(f \circ ?1) \circ (\text{trans } \circ [a,b]) \Rightarrow \text{ALL } f \circ a \text{ SI } (\text{DEF } \circ (\text{trans } \circ [a,b])),$$

puis par superposition avec E1, associativité de "o", on obtient la famille infinie de règles:

$$\begin{aligned} &\text{ALL}(f_1 \circ (f_2 \circ \dots \circ (f_n \circ ?1) \dots)) \circ (\text{trans } \circ [a,b]) \\ &\Rightarrow \text{ALL}(f_1 \circ (f_2 \circ \dots \circ f_n)) \circ a \text{ SI } (\text{DEF } \circ (\text{trans } \circ [a,b])). \end{aligned}$$

Mais E2:  $[f,g] \circ h \Rightarrow [f \circ h, g \circ h]$

se superpose sur les règles de cette famille engendrant des règles comme

$$\begin{aligned} &\text{ALL} [f_1 \circ [f_2 \circ [g_1 \circ ?1, g_2 \circ [x_1 \circ (x_2 \circ ?1), y \circ ?1]], z \circ ?1], k \circ ?1] \\ &\quad \circ (\text{trans } \circ [a,b]) \Rightarrow \text{ALL} [f_1 \circ [f_2 \circ [g_1, g_2 \circ [x_1 \circ x_2, y]], z], k] \circ a \\ &\quad \text{SI } (\text{DEF } \circ (\text{trans } \circ [a,b])). \end{aligned}$$

Nous voulons schématiser la famille infinie de règles  $(g_k \Rightarrow d_k)$  engendrée par ces superpositions.

Les structures sont

$f_1, f_1 \circ f_2, f_1 \circ (f_2 \circ f_3) \dots$

$[f_1, g_1], [f_1 \circ f_2, g_1 \circ g_2],$

$[f_1 \circ (f_2 \circ f_3), g_1 \circ (g_2 \circ g_3)] \dots$

$h_1 \circ [x_1, y_1], h_1 \circ (h_2 \circ [x_1, y_1]) \dots$

R0 doit contenir les règles E1 et E2, R0 est le système de réécriture appelé S-E1E2 dans la première partie et nous avons vérifié sa convergence.

Le squelette de la suite  $(g_k)$  des membres gauches est alors

$$\text{ALL}(F \circ ?1) \circ (\text{trans } \circ [a,b])$$

La méta-variable F a pour domaine l'ensemble des termes.

Deuxième partie

La méta-règle

$$\text{ALL}(F \circ ?1) \circ (\text{trans } \circ [a,b]) \Rightarrow \text{ALL } F \circ a,$$

~~~~~

modulo R0=S-E1E2 et si F a pour domaine l'ensemble des termes, schématise la famille infinie  $g_k \Rightarrow d_k$ .

Nous avons en effet, pour tout k,  $g_k \Rightarrow^{++} d_k$

Nous allons maintenant observer des exemples de divergence de REVE pris parmi ceux que nous avons décelés dans la première partie. Au cours de nos essais, nous avons observé ces cas, ce qui nous permet de schématiser des méta-règles qui seraient susceptibles de montrer que nous avons des systèmes de réécriture convergents avec les familles infinies de règles. Ces méta-règles pourront être utilisées et les résultats pourront être vérifiés avec une procédure de méta-complétion.

### 3. Des méta-règles solution de nos problèmes de divergence ?

Au lieu de rendre compte des expérimentations que nous avons faites avec REVE pour essayer d'observer les familles infinies que nous avons rencontrées, ce qui serait fastidieux sans apporter beaucoup d'information, nous ne donnerons que trois cas.

Par ces expérimentations, nous voyons quelles sont les familles infinies que nous devons schématiser et ainsi, nous pouvons regarder si une solution avec des méta-règles est possible.

Deuxième partie

### 3.1. La confluence du système de réécriture sur les primitives

Rappelons que le système S-PRIMITIVE présenté dans la première partie, n'est pas localement confluent, car nous avons été obligés de détruire les paires critiques obtenues par superposition de la règle

APPASS: append o [append o [a,b],c] => append o [a,append o [b,c]]

sur la règle:

APPTRANS: trans o [append o [a,b],append o [c,d]]

=> append o [trans o [a,c],trans o [b,d]]

SI (DEF o [trans o [a,c],trans o [b,d]]).

REVE réduit ces paires critiques avec la règle APPTRANS elle même, alors que la précondition n'est pas vérifiée, et on obtient des équations non valides que nous sommes tenues de détruire pour poursuivre notre expérience.

Cependant, il faut tenir compte de ces paires critiques pour que le système S-PRIMITIVE soit confluent.

On constate alors qu'une infinité de règles conditionnelles, comme par exemple

trans o [append o [a1,append o [b1,b2]],

append o [append o [c1,append o [c2,append o [d1,d2]]]]

=> append o [trans o [a1,append o [d1,d2]],

trans o [append o [b1,b2],append o [c1,c2]]]

SI (DEF o D) où D est le membre gauche, serait engendrée.

Deuxième partie

Nous schématisons cette famille infinie de règles conditionnelles par une méta-règle conditionnelle:

L'ensemble des structures est  
 ~~~~~

a1, append o [a1,a2], append o [a1,append o [a2,a3]],...

Nous considérons le système de réécriture convergent réduit à la règle APPASS.

Le domaine des méta-variables est l'ensemble des termes. La méta-règle s'écrit

trans o [append o [A,B],append o [C,D]]  
 ~~~~~

=> append o [trans o [A,C],trans o [B,D]]  
 ~~~~~

SI (DEF o [trans o [A,C],trans o [B,D]]),  
 ~~~~~

modulo APPASS.

En substituant la règle APPTRANS par cette méta-règle conditionnelle, le système S-PRIMITIVE est localement confluent.

### 3.2. Le système peut être complété avec l'axiome CONS

L'axiome

CONS: append o [L(hd o a),tl o a] => a SI (DEF o (tl o a))

engendre une famille infinie de règles avec les superpositions des règles

Deuxième partie

HDISTL:  $hd\ o\ (distl\ o\ [x,a]) \Rightarrow [x,hd\ o\ a]$

TLDISTL:  $tl\ o\ (distl\ o\ [x,a]) \Rightarrow distl\ o\ [x,tl\ o\ a]$

La suite des membres gauches (gi) est

$append\ o\ [L([x_1,hd\ o\ a]),distl\ o\ [x_1,tl\ o\ a]],$

$append\ o\ [L([x_1,[x_2,hd\ o\ a]]),distl\ o\ [x_1,distl\ o\ [x_2,tl\ o\ a]]],...$

Nous ne schématisons pas encore cette famille car les axiomes HTRANS et TLTRANS, HAPP et TLAPP, HIOTA et TLIOTA ainsi que HLIST et TLLIST vont aussi se superposer sur cette famille infinie donnant une famille infinie de règles dont les membres gauches sont généralisés par le terme initial

$append\ o\ [L(hd\ o\ a),tl\ o\ a]$ .

On doit considérer le système des règles

$S\text{-}HTL = \{HTRANS, TLTRANS, HDISTL, TLDISTL, HIOTA, TLIOTA, HAPP, TLAPP, HLIST, TLLIST\}$

qui est un sous-ensemble convergent dans S-PRIMITIVE.

L'ensemble des structures est

a,

$distl\ o\ [x,a], distl\ o\ [x_1, distl\ o\ [x_2, a]], ...$

$trans\ o\ [a,b], trans\ o\ [trans\ o\ [x,y], b], trans\ o\ [a, trans\ o\ [x,y]], ...$

$distl\ o\ [x_1, trans\ o\ [a,b]], distl\ o\ [x_1, distl\ o\ [x_2, trans\ o\ [a,b]]], ...$

Le domaine est l'ensemble des termes.

La méta règle CONS

$append\ o\ [L(hd\ o\ A), tl\ o\ A] \Rightarrow A\ SI\ (DEF\ o\ (tl\ o\ A)),$

Deuxième partie

modulo S-HTL, schématise la famille infinie.

REMARQUES

a) Signalons que l'on obtient, en résultat de la superposition de P2, une règle CONS'

$append\ o\ [L(hd), tl] \Rightarrow id\ SI\ (DEF\ o\ tl).$

Il n'y a pas d'autre superpositions d'une règle de S-PRIMITIVE sur une des règles de la famille.

b) Par contre, on observe que les règles de cette famille se superposent toutes sur APPASS et sur APPTRANS, ce qui donne des familles de règles que nous savons aussi schématiser. Nous n'alourdissons pas ce texte avec la description de ces schématisations.

c) Nous voyons aussi des superpositions qui engendrent des règles réduites par les règles de la famille.

Nous donnons en exemple les superpositions sur APPDISTL

APPDISTL:  $distl\ o\ [x, append\ o\ [a,b]]$

$\Rightarrow append\ o\ [distl\ o\ [x,a], distl\ o\ [x,b]]$

donnant la famille de paires critiques

$distl\ o\ [x,A] = append\ o\ [[x,hd\ o\ A], distl\ o\ [x,tl\ o\ A]] =_{R0} distl\ o\ [x,A].$

La famille infinie de règles correspondant à ces paires critiques est "méta-réduite".

Deuxième partie

Il est sans doute possible de compléter S-PRIMITIVE avec des  
 ~~~~~  
 méta-règles.  
 ~~~~~

### 3.3. Les axiomes ALL?1TRANS et ALL?2TRANS dans S-ALL

Nous introduisons les deux axiomes

ALL ?1 o (trans o [a,b]) => a et

ALL ?2 o (trans o [a,b]) => b SI (DEF o (trans o [a,b]))

dans le système S-ALL.

Pour chaque axiome, on observe une famille infinie de règles résultat des superpositions ALLALL' puis de E1 et E2, famille schématisable modulo S-E1E2 par:

ALL(F o ?1) o (trans o [a,b]) => ALL F o a et  
 ~~~~~

ALL(F o ?2) o (trans o [a,b]) => ALL F o b  
 ~~~~~

SI (DEF o (trans o [a,b])).

PROBLEME POSE PAR L'AXIOME E3

E3:  $x^- o y \Rightarrow x^-$  SI (DEF o [a,b])

donne une sorte de cas particulier, car il se superpose sur chaque règle de la famille, ce qui engendre une nouvelle famille de règles schématisables par

Deuxième partie

ALL(F o  $x^-$ ) o (trans o [a,b]) => ALL(F o  $x^-$ ) o a et

ALL(F o  $x^-$ ) o (trans o [a,b]) => ALL(F o  $x^-$ ) o b

SI (DEF o (trans o [a,b])).

On obtient par conséquent les paires critiques:

ALL(F o  $x^-$ ) o a = ALL(F o  $x^-$ ) o b SI (DEF o (trans o [a,b]))

qui ne sont pas orientables.

On peut (en utilisant la primitive "length" qui a pour résultat la longueur d'une séquence) introduire un axiome

ALL  $x^-$  o a = ALL  $x^-$  o (iota o (length o a))  
 ~~~~~

qui engendre une famille infinie que l'on peut schématiser par la méta-règle

ALL(F o  $x^-$ ) o a => ALL(F o  $x^-$ ) o (iota o (length o a)).  
 ~~~~~

Cette méta-règle réduit l'ensemble des paires critiques ci-dessus compte tenu de leurs préconditions.

SUR LES REGLES

FALL': trans o [ALL f o a, ALL g o a] => ALL[f,g] o a,

FALLg: trans o [a, ALL f o a] => ALL[id,f] o a

FALLd: trans o [ALL f o a, a] => ALL[f,id] o a.

Les règles des familles infinies précédentes se superposent donnant par

Deuxième partie

exemple la règle:

(1)  $\text{trans } o [ALL f \ o \ a, ALL g \ o \ b] \Rightarrow ALL[f \ o \ ?1, g \ o \ ?2] \ o \ (\text{trans } o [a, b])$

qui détruit la règle FALL'.

La superposition de ALLID ( $ALL \ id = id$ ) engendre alors les deux règles

(2)  $\text{trans } o [a, ALL f \ o \ b] \Rightarrow ALL[?1, f \ o \ ?2] \ o \ (\text{trans } o [a, b])$  et

(3)  $\text{trans } o [ALL f \ o \ a, b] \Rightarrow ALL[f \ o \ ?1, ?2] \ o \ (\text{trans } o [a, b])$

qui détruisent les règles FALLg et FALLd.

Avec P2, on obtient les deux règles

$\text{trans } o [a, ALL f] \Rightarrow ALL[?1, f \ o \ ?2] \ o \ (\text{trans } o [a, id])$  et

$\text{trans } o [ALL f, a] \Rightarrow ALL[f \ o \ ?1, ?2] \ o \ (\text{trans } o [id, a])$

qui détruisent les règles FALLg, FALLg', FALLd et FALLd'.

Cependant, les règles des familles infinies se superposent sur les règles ALLALL', FALL, (1), (2) et (3) qui se superposent de nouveau sur FALL, (1), (2) et (3) et avec les règles du système S-BASE engendrant alors des familles infinies de règles et de paires critiques dont nous extrayons des exemples significatifs.

exemple 1: La simplification

$ALL[f \ o \ (?1 \ o \ ?1), g \ o \ ?2] \ o \ (\text{trans } o ((\text{trans } o [a, b], c)))$

$\Rightarrow ALL[f \ o \ ?1, g \ o \ ?2] \ o \ (\text{trans } o [a, c])$

SI (DEF  $o \ (\text{trans } o [\text{trans } o [a, b], c])$ ).

exemple 2: La permutation des sélecteurs

$ALL[?2, ?1] \ o \ (\text{trans } o [a, b]) \Rightarrow \text{trans } o [b, a]$

$ALL[f \ o \ ?2, g \ o \ ?1] \ o \ (\text{trans } o [a, b]) = ALL[f \ o \ ?1, g \ o \ ?2] \ o \ (\text{trans } o [b, a])$

Deuxième partie

exemple 3: La modification des accès

$ALL[f \ o \ (?1 \ o \ ?1), g \ o \ (?2 \ o \ ?1, ?2)]$

$o \ (\text{trans } o [\text{trans } o [a, \text{trans } o [c, d]], e])$

$= ALL[f \ o \ ?1, g \ o \ [?1 \ o \ ?2, ?2 \ o \ ?2]]$

$o \ (\text{trans } o [a, \text{trans } o [\text{trans } o [c, d], e]]).$

exemple 4: Les répétitions

$ALL[k \ o \ [f \ o \ (?1 \ o \ ?1), g \ o \ (?2 \ o \ ?1)] , h \ o \ ?2]$

$o \ (\text{trans } o [\text{trans } o [a, b], a])$

$\Rightarrow ALL [ [k \ o \ (f \ o \ ?1), g \ o \ ?2] , h \ o \ ?1] \ o \ (\text{trans } o [a, b]).$

l'exemple 5: est un mélange des cas précédents

$ALL[f \ o \ (?2 \ o \ ?2), g \ o \ (?1 \ o \ ?1)]$

$o \ (\text{trans } o [\text{trans } o [a, c], \text{trans } o [a, b]])$

$\Rightarrow ALL[f \ o \ ?1, g \ o \ ?2] \ o \ (\text{trans } o [b, a]).$

Nous schématisons un ensemble de règles capables de détruire ces règles ou paires critiques.

Les structures sont

~~~~~

$f1, f1 \ o \ f2, f1 \ o \ (f2 \ o \ f3), \dots$

$[f1, f2], [[f1, f2], f3], [f1, [f2, f3]], \dots$

$k_1 \ o \ [f1, f2], k_1 \ o \ (k_2 \ o \ [f1, f2]), \dots$

$k_1 \ o \ [[f1, f2], f3], \dots$

Les méta-règles sont ( modulo S-BASE pour PERMUT et SIMPLIF)

~~~~~

Deuxième partie

PERMUT: ALL(F o [?2,?1]) o (trans o [a,b]) => ALL F o (trans o [b,a])  
 ~~~~~

SIMPLIF: ALL(F o [?1,?1 o ?2]) o (trans o [X,trans o [b,c]])  
 ~~~~~

=> ALL F o (trans o [X,b])  
 ~~~~~

et

ALL(F o [?1,?2 o ?2]) o (trans o [X,trans o [b,c]])  
 ~~~~~

=> ALL F o (trans o [X,c])  
 ~~~~~

SI (DEF o (trans o [X,trans o [b,c]])).

REPET: ALL f o (trans o [X,trans o [a,X]])  
 ~~~~~

=> ALL(f o [?1,[?2,?1]]) o (trans o [X,a])  
 ~~~~~

et

ALL f o (trans o [X,trans o [X,a]])  
 ~~~~~

=> ALL(f o [?1,[?1,?2]]) o (trans o [X,a]).  
 ~~~~~

REMARQUE:

Le domaine de la méta-variable X est l'ensemble des termes qui ne sont pas filtrables par "trans o x". Cela permet de n'appliquer SIMPLIF que sur les termes réduits par la règle

Deuxième partie

NORMAL: ALL f o (trans o [trans o [a,b],c])

=> ALL (f o [[?1,?1 o ?2],?2 o ?2]) o (trans o [a,trans o [b,c]]).

Les méta-règles SIMPLIF, REPET et PERMUT détruisent toutes les règles ou paires critiques de la famille.

a) Les règles de la famille des simplifications sont détruites par réécriture avec NORMAL et SIMPLIF.

Pour l'exemple 1, le membre droit

ALL[f o (?1 o ?1),g o ?2] o (trans o [trans o [a,b],c])

=>\*R avec R = NORMAL U S-BASE

ALL [f o ?1, g o (?2 o ?2)] o (trans o [a,trans o [b,c]])

==>>>SIMPLIF

ALL [f o ?1, g o ?2] o (trans o [a,c])

b) Les règles de la famille des permutations sont schématisées par PERMUT.

C'est le cas des deux règles de l'exemple 2.

c) Les règles de la famille des modifications d'accès sont réduites par la règle NORMAL.

Les deux membres de la règle de l'exemple 3 se réduisent en

ALL[f o ?1, g o [[?1 o ?2,?1 o (?2 o ?2)],?2 o (?2 o ?2)]]

o (trans o [a,trans o [c,trans o [d,e]]])

d) Les règles de la famille des répétitions sont détruites par réécriture avec NORMAL et REPET.

Pour l'exemple 4, le membre droit

Deuxième partie

ALL[k o [f o (?1 o ?1),g o (?2 o ?1)],h o ?2] o (trans o [trans o [a,b],a])  
 =>\*R avec R= NORMAL U S-BASE

ALL[k o [f o ?1,g o (?1 o ?2)],h o (?2 o ?2)] o (trans o [a,trans o [b,a]])  
 ==>>>REPET =>\*S-BASE

ALL[k o [f o ?1,g o ?2],h o ?1] o (trans o [a,b]).

REMARQUE

La méta-règle PERMUT provoque des chaînes infinies de réécriture. On a par exemple

ALL[f o ?2,g o ?1] o (trans o [a,b]) ==>>PERMUT

ALL[f o ?1,g o ?2] o (trans o [b,a]) ==>>PERMUT

ALL[f o ?2,g o ?1] o (trans o [a,b]) ==>>PERMUT

ALL[f o ?2,g o ?1] o (trans o [b,a]) .....

Il suffit de restreindre le domaine de la méta-variable F pour éviter cela.

Par exemple,

ALL[f o ?2,g o ?1] o (trans o [a,b]) =<sub>S-BASE</sub>

ALL([f o ?1,g o ?2] o [?2,?1]) o (trans o [a,b]),

$\sigma(F) = [f o ?1, g o ?2]$ , ce qui permet la méta-réécriture par PERMUT.

On a aussi, ALL[f o ?1,g o ?2] o (trans o [a,b]) =<sub>S-BASE</sub>

ALL([f o ?2,g o ?1] o [?2,?1]) o (trans o [a,b]),

$\sigma(F) = [f o ?1, g o ?2]$ , ce qui autorise encore l'application de PERMUT.

Mais, si on restreint le domaine de F en imposant que la feuille la plus à gauche de la structure soit différente de "?1", la méta-réécriture par PERMUT n'est plus possible. Cette restriction du domaine empêche le bouclage des méta-réécritures par la méta-règle PERMUT.

Deuxième partie

Le système de réécriture S-ALL doit alors pouvoir être  
 complété avec les axiomes ALL?1TRANS et ALL?2TRANS

#### 4. Conclusion

##### 4.1. Un espoir d'améliorer nos résultats en ajoutant des méta-règles

Nous avons étudié d'autres cas de divergence qu'il nous semble inutile de rapporter ici. Nous voyons déjà, en étudiant ces trois cas, que l'utilisation de méta-règles peut être une solution pour les cas de divergences que nous avons décelés dans la première partie. Il faut une procédure de méta-complétion pour vérifier la validité de ces méta-règles.

Pour les exemples de la première partie (exemple 7 expression (1), exemple 2 expression (2) et exemple 5 expression (1)), nous avons pu trouver les méta-règles à ajouter à nos systèmes de réécriture convergents pour obtenir la forme normale que nous désirons. C'est pourquoi nous sommes assez convaincus que l'utilisation des méta-règles nous offre la possibilité d'améliorer nos résultats.

Travailler de cette façon sur nos exemples nous a donné une autre idée. On peut peut-être trouver un moyen d'obtenir la forme normale d'un terme (dans le système complet infini) sans avoir forcément besoin de toutes les règles du système. A la limite, on peut vouloir faire de même à partir de l'ensemble des règles obtenues en orientant les axiomes.

P. GLOESS et J. P. LAURENT [Glo80] ont eu cette idée et présentent un

Deuxième partie

algorithme dit de "complétion dynamique". Leur but n'est pas de résoudre des divergences de la procédure de Knuth-Bendix mais d'obtenir une forme normale d'un terme donné en utilisant un système de réécriture moins important. Nous faisons quelques commentaires sur leur méthode.

#### 4.2. Méthode de complétion dynamique [Glo80]

Appelons  $R_i$  le système de règles,  $R_i$  n'est pas confluent.

Le terme est réécrit "de l'extérieur vers l'intérieur" et "de gauche à droite" (appelons cette stratégie de réécriture "LR"). La dérivation est conservée: C'est "l'histoire" de la réécriture.

Cette histoire permet de découvrir les réductions, faites sur le terme, qui correspondent à la superposition des membres gauches de deux règles "g2=>d2" sur "g1=>d1" à une occurrence  $u$  avec un unificateur  $G$ .

La règle " $G(g1[u<-d2]) \Rightarrow G(d1)$ " est alors ajoutée à  $R_i$ . Comme cette orientation ne tient pas compte d'un ordre de réduction quelconque, le processus risque de ne pas se terminer. On voit même apparaître de nouvelles variables dans le membre gauche. Le procédé semble très empirique.

Appelons  $R_1$  le système de règles obtenu et  $R$  le système complété par la procédure de Knuth-Bendix à partir de  $R_i$ . On a

$$R_i \subseteq R_1 \subseteq R.$$

Si  $t_0$  est la forme normale obtenue avec  $R_i$  et la stratégie LR,  $t_1$  le résultat de la complétion dynamique avec un terme  $t$  et  $t \downarrow_R$  la forme normale dans  $R$ , on a

Deuxième partie

$$t \Rightarrow^{*R_i} t_0 \Rightarrow^{*R_1} t_1 \Rightarrow^{*R} t \downarrow_R.$$

#### LIMITE DU PROCÉDÉ

On a  $t=t_0=t_1$  quand  $t$  est irréductible dans  $R_i$ . De plus il est rare que  $t_1$  soit égal à  $t \downarrow_R$ .

On peut voir aisément les limites de la méthode.

#### 4.3. Exploitation de cette idée

On peut systématiser l'algorithme de [Glo80] de la façon suivante

a) tenir compte d'un ordre comme dans l'algorithme de Knuth-Bendix pour orienter les paires critiques.

b) parcourir complètement l'arbre des dérivations du terme pour épuiser toutes les possibilités de découvrir des paires critiques capables d'engendrer de nouvelles règles.

On a alors les résultats suivants:

a) l'algorithme se termine

b) unicité de la forme normale du terme  $t$  dans le système  $R_1$ .

Mais la remarque que nous avons faite au paragraphe précédent sur la limite du procédé s'applique encore.

Pour dépasser cette limite, il faudrait pouvoir "retourner les règles" de  $R_i$ .

Deuxième partie

La situation qui nous intéresse est la suivante:

Nous avons un système de réécriture convergent  $R_i$  et un axiome qui provoque une divergence de la procédure de Knuth-Bendix. On veut obtenir la forme normale du terme pour le système complété avec la famille infinie de règles correspondant à la divergence, c'est à dire comme si l'on avait la méta-règle qui schématise la famille infinie.

Quand on réécrit avec une méta-règle, on réécrit modulo  $R_0$ . et comme " $R_0 \subseteq R_i$ ", cela revient à la possibilité de "retourner des règles de  $R_i$ ".

Il suffit d'appliquer le processus de méta-complétion à la réécriture du terme modulo  $R_0$  pour résoudre le problème que nous nous posons. L'idée est peut-être exploitable.

## 1. Le problème

### 1.1. Introduction

Nous avons jusqu'ici restreint la FP-algèbre en nous limitant aux sélecteurs " $?1$ " et " $?2$ " et en limitant l'arité de " $[_\dots_]$ " à 2. Ainsi, nous n'avons qu'un nombre fini d'axiomes, à partir desquels nous avons construit des systèmes de réécriture convergents.

Quand la FP-algèbre a une famille infinie d'axiomes correspondant aux arités 2,3,4,... de " $[_\dots_]$ ", comme par exemple

$$E2: [f1, f2] \circ h = [f1 \circ h, f2 \circ h],$$

$$[f1, f2, f3] \circ h = [f1 \circ h, f2 \circ h, f3 \circ h], \dots$$

nous n'avons gardé que le premier des axiomes de la famille qui correspond à l'arité 2.

On peut aussi avoir, pour une famille, un ensemble de règles pour chaque arité, comme par exemple la famille COND5 qui comprend

- un couple de règles pour l'arité 2

$$COND5: [p \rightarrow f; g, h1] \Rightarrow p \rightarrow [f, h1]; [g, h1],$$

$$COND6: [h1, p \rightarrow f; g] \Rightarrow p \rightarrow [h1, f]; [h1, g],$$

- un triplet de règles pour l'arité 3

$[p \rightarrow f; g, h1, h2] \Rightarrow p \rightarrow [f, h1, h2]; [g, h1, h2],$   
 $[h1, p \rightarrow f; g, h2] \Rightarrow p \rightarrow [h1, f, h2]; [h1, g, h2],$   
 $[h1, h2, p \rightarrow f; g] \Rightarrow p \rightarrow [h1, h2, f]; [h1, h2, g], \dots$

Nous n'avons gardé que le couple des deux premiers axiomes de la famille.

C'est la même chose quand la FP-algèbre a une famille infinie d'axiomes correspondant aux sélecteurs  $?1, ?2, \dots$ , et aux diverses arités  $1, 2, \dots$  de "[\_...\_]", comme par exemple la famille P3 qui comprend:

- un couple d'axiomes pour l'arité 2  
 $?1 \circ [f1, f2] = f1$  et  
 $?2 \circ [f1, f2] = f2$  SI (DEF  $\circ [f1, f2]$ )  
 - un triplet d'axiomes pour l'arité 3  
 $?1 \circ [f1, f2, f3] = f1,$   
 $?2 \circ [f1, f2, f3] = f2$  et  
 $?3 \circ [f1, f2, f3] = f3$  SI (DEF  $\circ [f1, f2, f3]$ ), ...

Nous n'avons gardé que les deux premiers axiomes de la famille qui concernent les sélecteurs " $?1$ " et " $?2$ " et l'arité 2 de "[\_...\_]".

Dans la pratique, il est exclus de considérer des ensembles infinis d'axiomes. Mais ces familles infinies d'axiomes correspondent à des familles infinies de règles que nous pouvons tenter de schématiser par des méta-règles comme pour les familles infinies de règles engendrées par la procédure de Knuth-Bendix qui diverge.

Le formalisme décrit au chapitre précédent est utilisable pour ces familles infinies. Nous ferons cette schématisation au paragraphe 2.

Le problème est tout de même plus simple que celui des divergences de la

procédure de Knuth-Bendix. Dans le cas de la FP-algèbre, il est assez facile de prouver que le système de réécriture infini, extension d'un des système de réécriture trouvés pour l'arité 2, est encore convergent.

Nous commençons par définir les extensions  $S_n$  de nos systèmes convergents  $S$  et nous regardons si les systèmes  $S_n$  sont convergents.

## 1.2. Une extension d'un système de réécriture $S$

dans l'ensemble des axiomes de la FP-algèbre (voir l'annexe I) est obtenue

a) pour les règles résultant de l'orientation des axiomes:

- en remplaçant un couple de règles de  $S$ , identiques au sélecteur près, par la famille infinie correspondante dans  $A$ .

- et, si la (ou les) équation(s) correspondante(s) est(sont) valide(s) pour une arité  $n$  quelconque, en remplaçant une règle (ou un groupe de règles) dont le membre gauche contient une occurrence de "[\_...\_]", par la famille infinie correspondante dans  $A$ ,

Nous dirons que l'extension est totale pour  $A$  quand on a tenu compte de toutes les familles infinies d'axiomes de  $A$ .

Exemple: Le système

$E1: (f \circ g) \circ h \Rightarrow f \circ (g \circ h)$   
 $E2: [f1, f2] \circ h \Rightarrow [f1 \circ h, f2 \circ h],$   
 $[f1, f2, f3] \circ h \Rightarrow [f1 \circ h, f2 \circ h, f3 \circ h], \dots$   
 $E3: x^- \circ y \Rightarrow x^- \text{ SI } (DEF \circ y)$   
 $P1: id \circ f \Rightarrow f$   
 $P2: f \circ id \Rightarrow f$   
 $P3: ?1 \circ [f1, f2] \Rightarrow f1 \text{ et}$   
 $?2 \circ [f1, f2] \Rightarrow f2 \text{ SI } (DEF \circ [f1, f2]),$   
 $?1 \circ [f1, f2, f3] \Rightarrow f1,$   
 $?2 \circ [f1, f2, f3] \Rightarrow f2 \text{ et}$   
 $?3 \circ [f1, f2, f3] \Rightarrow f3 \text{ SI } (DEF \circ [f1, f2, f3]), \dots$   
 est une extension totale pour A du système de réécriture S-BASE.

b) pour une nouvelle règle ajoutée par la procédure de complétion:

- dans le cas où cette nouvelle règle est le résultat d'une superposition d'une règle r1 avec une règle r2 remplaçable par une famille infinie F2 de règles, on ajoute la famille de règles issue des extensions des superpositions entre r1 et F2.

Exemple: Dans le système S-ALL, les règles

FALLg:  $\text{trans } \circ [id, ALL \ f] \Rightarrow ALL[id, f] \text{ et}$

FALLd:  $\text{trans } \circ [ALL \ f, id] \Rightarrow ALL[f, id]$

sont résultats de la superposition de

ALLID:  $ALL \ id \Rightarrow id \text{ sur}$

FALL:  $\text{trans } \circ [ALL \ f, ALL \ g] \Rightarrow ALL[f, g].$

La famille

Deuxième partie

$FALLg, FALLd,$   
 $\text{trans } \circ [id, ALL \ f1, ALL \ f2] \Rightarrow ALL[id, f1, f2],$   
 $\text{trans } \circ [ALL \ f1, id, ALL \ f3] \Rightarrow ALL[f1, id, f3],$   
 $\text{trans } \circ [ALL \ f1, ALL \ f2, id] \Rightarrow ALL[f1, f2, id],$   
 $\text{trans } \circ [ALL \ f, id, id] \Rightarrow ALL[f1, id, id],$   
 $\text{trans } \circ [id, ALL \ f, id] \Rightarrow ALL[id, f, id], \dots$   
 étend FALLg et FALLd.

- Il reste le cas où la nouvelle règle est issue des superpositions entre deux règles r1 et r2 s'étendant toutes deux dans des familles infinies F1 et F2. Ce serait par exemple le cas pour les deux familles infinies F1 et F2 issues des superpositions des axiomes

ALL ?1  $\circ (\text{trans } \circ [a, b]) \Rightarrow a \text{ et}$

ALL ?2  $\circ (\text{trans } \circ [a, b]) \Rightarrow b \text{ SI } (DEF \circ (\text{trans } \circ [a, b])).$

On ajouterait alors une famille infinie de règles issue des extensions des superpositions entre F1 et F2.

Nous ne rencontrons pas ce cas pour étendre, dans l'ensemble d'axiomes A, les systèmes de réécriture convergents trouvés dans la première partie.

Remarquons que dans le chapitre précédent, on partait d'une famille infinie de règles engendrées par la procédure de Knuth-Bendix qui diverge. Ici, ce sont les extensions de nos systèmes convergents qui servent de point de départ à notre schématisation.

Nous regardons si les les systèmes infinis, extensions des systèmes de réécriture convergents, trouvés dans la première partie sont convergents puis nous schématisons les familles infinies par des méta-règles.

Deuxième partie

## 2. Des systèmes de réécriture convergents

### 2.1. La propriété de confluence locale est-elle conservée?

Nous considérons les systèmes, extensions dans A (c'est à dire avec les axiomes de la FP-algèbre décrits dans l'annexe I) des systèmes de réécriture décrits dans l'annexe II.

Nous appelons n-règle la règle ou le groupe de règles correspondants à l'arité n.

Appelons n-système S le système de réécriture qui est le résultat du remplacement, dans un système de réécriture S, de ses 2-règles par les n-règles correspondantes.

Nous voulons prouver que les systèmes, extensions dans l'ensemble A d'axiomes de la FP-algèbre, sont localement confluents.

Nous indiquons seulement ici les grandes lignes de cette preuve. En fait, préciser la preuve au niveau de chaque règle n'est pas très difficile.

a) Nous regardons les superpositions d'une n-règle sur elle même.

Pour chaque n-règle, nous devons prouver, par récurrence sur n, que la n-règle se comporte comme la 2-règle correspondante (c'est à dire que les paires critiques d'une n-règle sont les extensions des paires critiques de la 2-règle).

Nos n-règles, sauf la famille infinie des règles F (voir l'annexe I),

extension des axiomes de fusion et compositions sur "\*" ( $F^*, FALL^*, FALL^*S, F^{**}$  et  $F^{**}S$ ), possèdent cette propriété.

On a par exemple deux 3-règles de F:

$$\text{trans } o [ALL \text{ } g \text{ } o (*f \text{ } o [x,a]), *f \text{ } o [x,a], *f \text{ } o [x,a]] \Rightarrow W1$$

$$W1 = * [g \text{ } o [f \text{ } o [?2o?1,?2], f \text{ } o [?2o?1,?2], f \text{ } o [?3o?1,?2]] \text{ } o [[g \text{ } o x,x,x], a]$$

$$\text{et trans } o [ALL \text{ } g \text{ } o (*f \text{ } o [x,a]), *f \text{ } o [x,a], *f \text{ } o [x,a]] \Rightarrow W2$$

$$W2 = * [g \text{ } o [f \text{ } o [?3o?1,?2], f \text{ } o [?2o?1,?2], f \text{ } o [?3o?1,?2]] \text{ } o [[g \text{ } o x,x,x], a].$$

qui ne diffèrent que par le sous-terme signalé en caractères gras. Une conséquence équationnelle est  $W1 = W2$ . A partir de la famille F, on obtient ainsi une famille infinie d'équations. La solution existe, c'est la même que celle que nous avons exposée dans le chapitre précédent à propos de la solution par méta-règles des divergences avec les règles de fusion dans le système S-ALL. Nous ne tiendrons compte de F que lorsque nous pourrons valider les méta-règles à l'aide de la procédure de méta-complétion.

Nous nous limitons à une extension de nos systèmes de réécriture convergents dans l'ensemble d'axiomes A-{F}.

b) Nous pouvons prouver, par récurrence sur l'arité n de "[...]", que les paires critiques entre une n-règle et les autres règles du n-système sont les extensions des paires critiques trouvées pour le 2-système. C'est vrai pour les extensions de nos systèmes de réécriture.

A ce stade de la vérification, on peut dire que: pour une extension totale dans A-{F}, les n-systèmes sont localement confluents.

c) De plus, pour tout  $i \neq j$ , une i-règle R ne se superpose pas sur la j-règle R et une i-règle R1 ne se superpose pas sur une j-règle R2.

On peut donc conclure que:

les systèmes extensions, dans l'ensemble d'axiomes  $A-\{F\}$  de la FP-algèbre, de nos systèmes convergents sont localement confluents.

## 2.2. La propriété de terminaison finie est conservée.

Une  $n$ -règle s'oriente de la même façon qu'une 2-règle quel que soit l'ordre choisi.

On peut donc conclure que:

les systèmes de réécriture extensions dans l'ensemble d'axiomes  $A-\{F\}$  de la FP-algèbre des systèmes de réécriture convergents trouvés dans la première partie sont convergents.

Nous allons maintenant schématiser par des méta-règles les familles infinies de règles incluses dans les systèmes étendus.

## 3. Schématization des familles infinies de règles

Nous utilisons le formalisme des méta-règles (voir le chapitre précédent) pour schématiser les familles infinies de règles de nos systèmes étendus.

Nous considérons chacun de nos systèmes de réécriture convergents trouvés dans la première partie. Nous construisons les systèmes extensions totales dans l'ensemble d'axiomes  $A-\{F\}$ . Puis nous schématisons chaque famille infinie de règles.

Deuxième partie

Le résultat de ce travail figure dans l'annexe III. Nous nous contentons dans ce paragraphe de donner des exemples significatifs de schématisation.

Les schématisations que nous avons faites au chapitre précédent concernent des familles infinies de règles engendrées par l'algorithme de Knuth-Bendix qui diverge. Les familles infinies de règles ne sont pas de même nature et leur schématisation est particulière. Nous avons 2 sortes de familles infinies à schématiser, celles qui sont engendrées par les arités 2,3,... de "[...]" et celles qui dépendent des sélecteurs ?1,?2,...

Pour toutes les définitions, résultats et notations qui concernent le processus de schématisation et les méta-règles, il faut se reporter au chapitre précédent.

## 3.1. Schématization des familles infinies engendrées par "[...]"

### Généralités

● Un premier problème est de généraliser les suites infinies

$[t_1, t_2],$   
 $[t_1, t_2, t_3],$   
 $[t_1, t_2, t_3, t_4], \dots$

Nous voulons considérer un nombre fini de symboles "[...]" avec une arité fixe, qui peut en particulier être l'arité 2. La liste des termes " $t_2, t_3, t_4, \dots$ " peut être schématisée par un terme

$$t_2.(t_3.(t_4. \dots$$

Il suffit d'ajouter un nouveau symbole "." qui représente une opération

Deuxième partie

associative.

Il existe une bijection  $\theta$  entre la suite ci-dessus et la suite

$[t_1, t_2],$   
 $[t_1, t_2, t_3],$   
 $[t_1, t_2, (t_3, t_4)], \dots$

Le système de réécriture  $RO$  contient alors l'axiome d'associativité de  
 ",":

ASS:  $(x.y).z \Rightarrow x.(y.z)$   
 ~~~~~

Les structures sont

$x_1, x_1.x_2, x_1.(x_2.x_3), \dots$

Le généralisé modulo  $RO$  de la suite ci-dessus est alors "[x,y]".

La méta-variable  $Y$  a pour domaine  $DOM$ , sous-ensemble de l'algèbre  
 $T(SU\{.,V\})$ , ( $S$  est ici l'ensemble des symboles de la FP-algèbre) suivant:

$DOM = \{t_1, t_1.t_2, t_1.(t_2.t_3), \dots\}$

où  $t_1, t_2, t_3, \dots$  sont des éléments de  $T(S,V)$ .

On peut décider si un terme appartient à  $DOM$  défini par

$t \in T(S,V)$  implique  $t \in DOM$  et

$t_1 \in DOM$  et  $t_2 \in DOM$  implique  $t_1.t_2 \in DOM$ .

⊗ Le deuxième problème est de définir  $RO$  pour que, si la famille  
 infinie de règles est  $(g_k \Rightarrow d_k)$  et la méta-règle est  $MR$ , on ait, pour tout  
 $k$

Deuxième partie

$g_k \Rightarrow \dots \Rightarrow^+ MR d_k$ .

Nous distinguons deux cas selon l'origine de la famille infinie.

a) la famille infinie de règles provient de l'orientation d'une famille  
 infinie d'axiomes.

Considérons l'exemple de la famille  $E2$

$E2: [f1, f2] o h \Rightarrow [f1 o h, f2 o h],$

$[f1, f2, f3] o h \Rightarrow [f1 o h, f2 o h, f3 o h], \dots$

qui, avec  $\theta$  devient

$E2: [f1, f2] o h \Rightarrow [f1 o h, f2 o h],$

$[f1, f2, f3] o h \Rightarrow [f1 o h, (f2 o h).(f3 o h)], \dots$

Si le système de réécriture  $RO$  est composé de  $ASS$  et de l'axiome

$DIST: (f.g) o h \Rightarrow (f o h).(g o h),$   
 ~~~~~

il est convergent et la méta-règle

$E2: [f, G] o h \Rightarrow [f o h, G o h]$

où  $G$  a pour domaine  $DOM$ , schématise la famille  $E2$ .

La plupart des familles infinies de règles se schématise comme  $E2$ .

⊗ Le squelette du membre gauche de la méta-règle est le membre gauche de  
 la 2-règle de la famille.

⊗ Pour l'occurrence  $u$  de  $[\_]$  origine de l'infinité, on place une méta-  
 variable de domaine  $DOM$  à l'occurrence  $u_2$ .

⊗  $RO$  est constitué de  $ASS$  et des règles qui permettent d'avoir

Deuxième partie

$$g_k \Rightarrow^{+} MR d_k.$$

• On vérifie que RO est convergent en utilisant REVE si c'est nécessaire.

EXEMPLES:

1) La règle FALL:

$trans\ o\ [ALL\ f, ALL\ g] \Rightarrow ALL[f, g]$

est étendue avec la famille infinie

$trans\ o\ [ALL\ f_1, ALL\ f_2, ALL\ f_3, \dots, ALL\ f_n] \Rightarrow ALL[f_1, f_2, \dots, f_n]$

en bijection avec

$trans\ o\ [ALL\ f_1, ALL\ f_2.(ALL\ f_3.(...ALL\ f_n)...)]$   
 $\Rightarrow ALL[f_1, f_2.(f_3.(...f_n)...)]$

et est schématisée par la méta-règle

$trans\ o\ [ALL\ f, ALL\ G] \Rightarrow ALL[f, G].$

G a pour domaine DOM.

$RO = \{ASS, ALL\}$  avec  $ALL: ALL(x.y) \Rightarrow ALL\ x . ALL\ y.$

2) La 2- règle HTRANS:

$hd\ o\ (trans\ o\ [a, b]) \Rightarrow [hd\ o\ a, hd\ o\ b] \text{ SI } (DEF\ o\ (trans\ o\ [a, b]))$

est étendue avec la famille infinie

$hd\ o\ (trans\ o\ [a_1, a_2, \dots, a_n]) \Rightarrow [hd\ o\ a_1, hd\ o\ a_2, \dots, hd\ o\ a_n]$

$SI\ (DEF\ o\ (trans\ o\ [a_1, a_2, \dots, a_n]))$

Deuxième partie

qui, avec  $\theta$ , devient

$hd\ o\ (trans\ o\ [a_1, a_2.(a_3...a_n)])$

$\Rightarrow [hd\ o\ a_1, (hd\ o\ a_2).((hd\ o\ a_3)... (hd\ o\ a_n))]$

et est schématisée par la méta-règle HTRANS:

$hd\ o\ [a, B] \Rightarrow [hd\ o\ a, hd\ o\ B] \text{ SI } (DEF\ o\ (trans\ o\ [a, B]))$

B a pour domaine DOM.

$RO = \{ASS, HD\}$  avec  $HD: hd\ o\ (x.y) \Rightarrow (hd\ o\ x).(hd\ o\ y).$

3) La 2-règle APPTRANS:

$trans\ o\ [append\ o\ [a, b], append\ o\ [c, d]]$

$\Rightarrow append\ o\ [trans\ o\ [a, c], trans\ o\ [b, d]]$

$SI\ (DEF\ o\ [trans\ o\ [a, c], trans\ o\ [b, d]])$

est étendue, après application de  $\theta$ , avec la famille infinie

$trans\ o\ [append\ o\ [a_1, b_1], (append\ o\ [a_2, b_2]).((... (append\ o\ [a_n, b_n])))]$

$\Rightarrow append\ o\ [trans\ o\ [a_1, a_2.(...a_n)], trans\ o\ [b_1, b_2.(...b_n)]]$

$SI\ (DEF\ o\ [trans\ o\ [a_1, a_2.(...a_n)], trans\ o\ [b_1, b_2.(...b_n)]])$

et est schématisée par la méta-règle

$trans\ o\ [append\ o\ [a, b], append\ o\ [C, D]]$

$\Rightarrow append\ o\ [trans\ o\ [a, C], trans\ o\ [b, D]]$

$SI\ (DEF\ o\ [trans\ o\ [a, C], trans\ o\ [b, D]])$

C et D ont pour domaine DOM.

$RO = \{ASS, APP\}$  avec APP:

Deuxième partie

$\text{append } o [x,y,u,v] \Rightarrow (\text{append } o [x,u]).(\text{append } o [y,v]).$

#### 4) La 2-règle

$\text{COND5: } [p \rightarrow f; g, h] \Rightarrow p \rightarrow [f, h]; [g, h]$

$\text{COND6: } [h, p \rightarrow f; g] \Rightarrow p \rightarrow [h, f]; [h, g]$

est étendue avec la famille infinie

$[h_1, \dots, h_{i-1}, p \rightarrow f; g, h_{i+1}, \dots, h_n]$

$\Rightarrow p \rightarrow [h_1, \dots, h_{i-1}, f, h_{i+1}, \dots, h_n]; [h_1, \dots, h_{i-1}, g, h_{i+1}, \dots, h_n]$

qui comprend les suites

$[p \rightarrow f; g, h_1, \dots, h_n] \Rightarrow p \rightarrow [f, h_1, \dots, h_n]; [g, h_1, \dots, h_n]$  et

$[h_1, \dots, h_n, p \rightarrow f; g] \Rightarrow p \rightarrow [h_1, \dots, h_n, f]; [h_1, \dots, h_n, g].$

Introduisons un nouveau symbole "NIL" qui nous permet de considérer la liste vide. Une bijection  $\phi$  donne la suite infinie

$[(h_1(\dots h_{i-1}))((p \rightarrow f; g).(h_{i+1}(\dots h_n)))] \Rightarrow$

$p \rightarrow [(h_1(\dots h_{i-1})).(f.(h_{i+1}(\dots h_n)))]; [(h_1(\dots h_{i-1})).(g.(h_{i+1}(\dots h_n)))]$

qui comprend les suites

$[\text{NIL}((p \rightarrow f; g).(h_1 \dots h_n))] \Rightarrow p \rightarrow [\text{NIL}.(f.(h_1 \dots h_n))]; [\text{NIL}.(g.(h_1 \dots h_n))]$  et

$[h_1(\dots h_n)((p \rightarrow f; g).\text{NIL})] \Rightarrow p \rightarrow [h_1(\dots h_n).(f.\text{NIL})]; [h_1(\dots h_n).(g.\text{NIL})].$

Par la bijection  $\phi$ , "[\_]" devient un symbole d'arité 1.

Les structures sont NIL,  $h_1$ ,  $h_1.h_2$ ,  $h_1.(h_2.h_3) \dots$

La famille infinie est alors schématisée par la méta-règle

$[H_1.(p \rightarrow f; g.H_2)] \Rightarrow p \rightarrow [H_1.(f.H_2)]; [H_1.(g.H_2)].$

$R_0 = \{\text{ASS}, \text{NILg}, \text{NILd}\}$  avec  $\text{NILd: } x.\text{NIL} \Rightarrow x$  et  $\text{NILg: } \text{NIL}.x \Rightarrow x$ .

Le domaine de  $H_1$  et  $H_2$  est DOMV qui est défini par

Deuxième partie

- NIL  $\in$  DOMV

-  $t \in T(S,V)$  implique  $t \in \text{DOMV}$

-  $t_1 \in \text{DOMV}$  et  $t_2 \in \text{DOMV}$  implique  $t_1.t_2 \in \text{DOMV}$ .

b) la famille infinie de règles est engendrée par complétion.

Dans ce cas, nous pouvons trouver  $R_0$  en tenant compte des règles  $r$  qui, superposées avec les règles de la famille infinie  $F_1$  sont à l'origine de cette nouvelle famille infinie  $F_2$ . Nous donnons deux exemples.

a) Exemple 1: extension de FALL':

$\text{FALL': } \text{trans } o [ALL f \circ h, ALL g \circ h] \Rightarrow ALL[f, g] \circ h$

est résultat de la superposition de

$\text{FALL: } \text{trans } o [ALL f, ALL g] \text{ sur } E_1: (f \circ g) \circ h \Rightarrow f \circ (g \circ h).$

La famille infinie  $F'$  qui étend  $\text{FALL'}$  après application de  $\theta$  devient

$\text{trans } o [ALL f \circ h, (ALL g_1 \circ h).((ALL g_2 \circ h) \dots)]$

$\Rightarrow ALL[f, g_1.(g_2 \dots)] \circ h$

et est schématisée par la méta-règle:

$\text{trans } o [ALL f \circ h, ALL G \circ h] \Rightarrow ALL[f, G] \circ h.$

$G$  a pour domaine DOM.

Le système  $R_0$  est  $\{\text{ASS}, \text{DIST}, \text{ALL}\}$  avec

$\text{ALL: } ALL(x.y) \Rightarrow (ALL x).(ALL y).$

Les règles ASS et ALL sont associées à la méta-règle schématisant l'extension de FALL. ALL' est le résultat de la superposition de ALL sur

Deuxième partie

E1.

b) Exemple 2: extension de FALLg', FALLd' et TALL:

FALLg': trans o [h, ALL f o h] => ALL[id, f] o h,

FALLd': trans o [ALL f o h, h] => ALL[f, id] o h et

TALL: trans o [h, h] => ALL[id, id] o h,

sont issues des superpositions de ALLID: ALL id => id

sur FALL': trans o [ALL f o h, ALL g o h] => ALL[f, g] o h.

Les familles infinies qui étendent ces 3 règles peuvent être schématisées par la méta-règle:

trans o [ALL F o h, ALL G o h] => ALL[F, G] o h.

RO = {ASS, DIST, ALL, ALLID} qui est convergent. F et G ont pour domaine DOM.

Remarquons que cette méta-règle schématise aussi la famille infinie F' qui étend FALL'.

Nous pouvons aussi constater que, du point de vue d'une méta-complétion, cette méta-règle "méta-réduit" la méta-règle qui schématise F'.

### 3.2. Schématisation de familles infinies de règles incluant des sélecteurs

#### L'extension des sélecteurs

Dans la première partie, nous n'utilisions que les sélecteurs "1" et "2" et comme l'arité de "[\_]" était limitée à 2, les sélecteurs ne pouvaient que s'appliquer à un couple. C'est pour cela que nous avons pu introduire l'axiome ID

Deuxième partie

[1, 2] = id.

Pour étendre cet axiome, il faut pouvoir distinguer les sélecteurs selon la taille du n-uplet auxquels ils s'appliquent.

R. B. KIEBURZ et G. S. GIVLER [Giv84] font aussi cette distinction. Les sélecteurs sont alors notés:

$1^1, 1^2, 1^3, \dots, 2^1, 2^2, 2^3, \dots$

et on a

$1^3: [2, 3, 4] = 2$ , mais  $2^2: [2, 3, 4] = !$  et  $2^5: [2, 3, 4] = !$ .

C'est un moyen pour distinguer les n-uplets en FP et donc un début d'introduction de la notion de type absente de ce langage.

#### La famille infinie d'axiomes ID

$[1^n, 2^n, \dots, n^n] = id$

peut maintenant être introduite dans l'ensemble A d'axiomes et nous étendons les systèmes de réécriture compte tenu de cette famille infinie d'axiomes.

En particulier la règle ID

$[1^2, 2^2] => id$

s'étend en une famille infinie de règles

$[1^n, 2^n, \dots, n^n] => id$

qui est transformée par  $\theta$  en

Deuxième partie

$[?1^n.((2^n)\dots(n^n))]$ .

Les structures et le domaine DOMS peuvent être l'ensemble infini des termes clos:

$?1^2. ?2^2, ?1^3.(?2^3. ?3^3), \dots$

DOMS est décidable par:

$?1^2. ?2^2 \in \text{DOMS},$

$x.(?1^n. ?(i+1)^n) \in \text{DOMS}$  si  $E(x). ?i^{(n-1)} \in \text{DOMS}$

où  $E(x)$  est une opération sur les listes de sélecteurs qui décrémente de 1 tous les exposants.

$RO = \{\text{ASS}\}.$

La méta-règle s'écrit alors

$[I] \Rightarrow \text{id.}$  I a pour domaine DOMS.

#### Les règles P3

$?1 \circ [f, g] \Rightarrow f$  et

$?2 \circ [f, g] \Rightarrow g$  SI  $(\text{DEF} \circ [f, g])$

s'étendent dans la famille infinie de règles

$?i^n \circ [f_1, f_2, \dots, f_i, \dots, f_n] \Rightarrow f_i$  SI  $(\text{DEF} \circ [f_1, f_2, \dots, f_n])$

pour  $i$  et  $n$  entiers strictement positifs tels que  $i \leq n$  et  $n > 1$ .

Intuitivement, nous allons, par méta-réécriture, suivre la liste  $f_1, f_2, \dots, f_n$  pour extraire  $f_i$ .

Nous avons maintenant besoin d'une opération qui ajoute un terme à gauche d'une liste de termes, opération que nous notons  $':$ . Il existe une

Deuxième partie

bijection  $\psi$  qui, appliquée à la famille ci-dessus, donne la suite infinie

$?i^n \circ [f_1:(f_2:\dots f_i:(\dots f_n))] \Rightarrow f_i$

Le squelette est alors:

$?1^N \circ [f':G].$

Les structures se confondent avec les domaines et sont les entiers strictement positifs pour la méta-variable I et les entiers supérieurs à 1 pour la méta-variable N.

Pour G, les structures sont les listes

$g_1, g_1:g_2, g_1:(g_2:g_3), \dots$

$RO = \{\text{VG}\}$  avec VG:  $x:NIL \Rightarrow x$ .

La méta-variable G a pour domaine LG les listes de termes construites avec l'opération  $':$ . LG est décidable, c'est l'ensemble des termes définis par

-  $NIL \in LG$

-  $x':z \in LG$  si  $x \in T(S, V)$  et  $z \in LG$ .

Nous introduisons aussi l'opération  $(-1)$  sur les entiers strictement positifs.

Avec les méta-règles

(1)  $?I^N \circ [f':G] \Rightarrow ?(I-1)^{(N-1)} [G]$  et

(2)  $?1^N \circ [f':G] \Rightarrow f$  SI  $(\text{DEF} \circ [f':G])$ ,

nous avons

Deuxième partie

$\exists i \sup n \circ [f1:(f2:\dots fn)] \Rightarrow \dots (1) \Rightarrow \dots (2) fi$

pour tous les couples d'entiers  $i$  et  $n$  strictement positifs tels que  $i \leq n$ .

La règle F\*

$F*: \text{trans } \circ [*f \circ [x,a], *g \circ [y,a]]$

$\Rightarrow [*f \circ [\exists 1 \circ \exists 2, \exists 2], g \circ [\exists 2 \circ \exists 1, \exists 2]] \circ [[x,y],a]$

s'étend en une famille infinie

$\text{trans } \circ [*f1 \circ [x_1,a], *f2 \circ [x_2,a], \dots, *fn \circ [x_n,a]] \Rightarrow$   
 $*[f1 \circ [\exists 1^n \circ \exists 2^n, \exists 2^n], f2 \circ [\exists 2^n \circ \exists 1^n, \exists 2^n]$   
 $\dots, fn \circ [\exists n^n \circ \exists 1^n, \exists 2^n]]$   
 $\circ [[x_1, x_2, \dots, x_n], a].$

Nous voulons, cette fois, examiner les listes de droite à gauche et nous introduisons l'opération qui ajoute un élément à droite d'une liste que nous notons "i". Une bijection  $\psi'$  donne, par exemple pour la suite des membres gauches:

$\text{trans } \circ [(*f1 \circ [x_1,a] \dots *fn-1 \circ [x_{n-1},a]) : *fn \circ [x_n,a]].$

Le squelette est

$\text{trans } \circ [( *F \circ [X,a]) : (*g \circ [y,a])].$

Pour  $F$  et  $X$ , les structures sont

$x_1, x_1 \cdot x_2, (x_1 \cdot x_2) \cdot x_3, \dots$

$RO$  contient les axiomes  $\{VD, **, *NIL, DL, DLNIL\}$  avec

Deuxième partie

$VD: NIL : x \Rightarrow x,$

$** : *(x:y) \Rightarrow *x : *y, *NIL : *NIL \Rightarrow NIL,$

$DL: x : y \circ [u:v,h] \Rightarrow (x \circ [u,h]) : (y \circ [u,h])$  et

$DLNIL: NIL \circ [NIL,h] \Rightarrow NIL.$

Pour obtenir la suite  $(d_k)$  des membres droits, on introduit encore deux opérations:

- opération (que l'on note  $L(x)$ ) qui, appliquée à la liste  $x$ :

$\langle \dots \langle (x_1 : x_2) : x_3 \rangle \dots \rangle : x_n$

a pour résultat  $n+1$  et

- une opération (que l'on note  $K(p,n)$ ) ayant pour résultat :

$\langle \dots (\exists 1^n \circ \exists 1^n : \exists 2^n \circ \exists 1^n) \dots \rangle : \exists p^n \circ \exists 1^n.$

On met dans  $RO$  les axiomes

- de  $L$

$L1: L(NIL) = 1,$

$L2: L(x:y) \Rightarrow L(x) + 1,$  et

- de  $K$

$K1: K(1,n) \Rightarrow \exists 1^n \circ \exists 1^n,$

$K2: K(p+1,n) \Rightarrow K(p,n) : (\exists p^n \circ \exists 1^n).$

La méta-règle s'écrit

$\text{trans } \circ [( *F \circ [X,a]) : (*g \circ [y,a])]$

$\Rightarrow [*[ \langle F \circ [K(L(F)-1, L(F)), \exists 2^{L(F)} \rangle ] :$

$\langle g \circ [\exists n^{L(F)} \circ \exists 1^{L(F)}, \exists 2^{L(F)} \rangle ] \circ [[x:y],a].$

Deuxième partie

#### 4. Conclusion

Nous pouvons schématiser par des méta-règles les familles infinies de règles des systèmes de réécriture, extensions dans l'ensemble A (voir l'annexe I) d'axiomes de la FP-algèbre, des systèmes de réécriture convergents de l'annexe II. Nous obtenons ainsi les systèmes de réécriture convergents qui figurent dans l'annexe III.

La notion de système de réécriture, extension dans un ensemble infini d'axiomes, nous permet uniquement de trouver, à partir des systèmes de réécriture convergents décrits dans l'annexe II et de l'ensemble infini d'axiomes défini dans l'annexe I, des systèmes de réécriture infinis convergents équivalents qui tiennent compte des ensembles infinis d'axiomes. Si nous possédions une implémentation d'une procédure de méta-complétion [Kir85], cette notion serait inutile.

Dans ce cas, il suffirait de savoir schématiser les familles infinies d'axiomes et on utiliserait la procédure de méta-complétion, dès le départ, avec comme données les méta-règles résultats de cette schématisation.

Contrairement à la schématisation des familles infinies de règles engendrées par des divergences de la procédure de Knuth-Bendix, trouver une méta-règle pour schématiser des familles infinies de règles provenant des familles infinies d'axiomes de la FP-algèbre, rend nécessaire l'introduction de nouveaux symboles.

Les nouvelles opérations à introduire pour les familles infinies de règles dues à l'ensemble infini de symboles correspondant:

- à l'arité quelconque de "[...]" qui sont des opérations sur les listes de termes  $t_1, t_2, \dots, t_n$  arguments de "[...]". Nous avons introduit les symboles "." (concaténation de deux listes), ":" (adjonction d'un élément à gauche), ";" (adjonction d'un élément à droite) et "NIL" (liste vide).

Les listes  $t_1, t_2, \dots, t_n$  sont mises en bijection avec les éléments du domaine des méta-variables.

- à l'ensemble infini des sélecteurs

$?1^n, ?2^n, \dots$  pour  $n=1, 2, \dots$

qui sont des opérations sur les entiers  $1, 2, \dots$

Les sélecteurs sont mis en bijection avec les entiers positifs stricts.

La FP-algèbre est formée d'axiomes conditionnels. Nous avons signalé au paragraphe 2 du chapitre II de la première partie comment un ensemble d'équations conditionnelles permet de définir une congruence syntaxique. Deux termes sont égaux si l'on peut passer de l'un à l'autre par une chaîne de remplacements d'un égal par un égal, chaque remplacement n'étant permis que si la précondition associée à l'axiome est vérifiée.

Nous avons associé aux axiomes conditionnels des règles de réécriture conditionnelles que nous écrivons sous la forme

$g \Rightarrow d$  SI  $p$  où la précondition  $p$  est un prédicat.

Pour pouvoir appliquer la règle  $g \Rightarrow d$ , il faut que le prédicat  $p$  soit vérifié.

La plupart de nos règles conditionnelles ont des préconditions de la forme "DEF o t" ce qui signifie que l'application "t:x" de la fonction  $t$  à tout objet  $x$  est différente de "I". La fonction  $t$  code un programme qui se termine, aussi, vérifier que  $t$  est défini, revient à vérifier que l'expression fonctionnelle de  $t$  est bien formée, ce qui va nous conduire à typer les expressions fonctionnelles.

Par ailleurs, il est utile, pour tenir compte de théories particulières, de considérer des règles de la forme

$m = n \mid - g \Rightarrow d$ ,

règles que nous avons introduites au chapitre II de la première partie.

Dans ce cas, si  $A$  est l'ensemble des axiomes, vérifier l'égalité  $m =_A n$  est nécessaire pour appliquer la règle  $g \Rightarrow d$ . D'une façon générale, une règle conditionnelle  $r$  est formée de deux parties les prémisses  $\text{prem}(r)$  et la conclusion  $g \Rightarrow d$ . Ces règles sont donc des règles conditionnelles.

Pour les règles conditionnelles, on doit définir une relation de réécriture qui nécessite d'évaluer les prémisses  $\text{prem}(r)$  pour pouvoir appliquer la conclusion  $g \Rightarrow d$ . Si l'on ne donne pas de restrictions sur les règles, on ne sait pas décider si un terme est en forme normale. Le problème se pose ensuite de savoir décider de la confluence de ces systèmes de réécriture.

Dans la littérature, diverses restrictions sur les prémisses sont proposées pour rendre la réécriture conditionnelle décidable. Une première approche ([Rem82], [Rem84]) propose d'introduire une hiérarchie entre les règles, les prémisses étant évaluées au niveau le plus bas de la hiérarchie. Une autre approche, moins restrictive, est proposée par [Kap84]. Les prémisses sont de la forme  $m = n$  et si on a un ordre de simplification (voir première partie chapitre III) tel que  $g > d$ ,  $g > m$  et  $g > n$ , la recherche d'une forme normale d'un terme est décidable.

Des solutions souvent partielles de procédures dérivées de la procédure de Knuth-Bendix sont aussi proposées accompagnées de restrictions sur les règles.

Nous allons regarder, si, pour chacune des formes de prémisses des axiomes de la FP-algèbre, la réécriture peut être décidable. Ce chapitre ne fait que préciser les problèmes ouverts mais tout reste à faire dans ce domaine.

### 1. Règles à prémisses "DEF o t": un problème de contrôle de type.

Nous ne nous intéressons qu'à la classe des expressions définies, c'est à dire des expressions  $e$  telles que "DEF o  $e$ ". La FP-algèbre est en principe non typée. Nous allons typer la FP-algèbre afin d'être en mesure de décider qu'un terme est défini quand il est "bien typé".

#### 1.1. Définition des types pour une procédure de "type checking"

Nous allons définir la FP-algèbre comme une algèbre typée en associant une "sorte" ou un type à chacun des symboles de la FP-algèbre.

##### a) Les formes fonctionnelles

$FUNC = \{ \_o, \_, \_-\_;\_, ALL\_, *\_, WH(\_) \}$ .

"o" est une opération de profil  $F \times F \rightarrow F$  où  $F$  désigne le type fonction et  $\times$  l'opération produit cartésien sur les types. Mais cette définition n'est pas suffisamment précise car elle n'indique pas que la coarité d'une fonction est égale à l'arité de l'autre. Le type fonction est polymorphe. Nous introduisons des variables de type que nous notons par des lettres grecques  $\alpha, \beta, \dots$  et décrivons le profil de "o" par

$o: \beta \rightarrow \xi \times \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \xi$

Le polymorphisme des types est ici plus simple que celui que l'on trouve dans les autres langages fonctionnels comme par exemple en ML [Mil78]. Les fonctionnelles comme "o", "ALL", ... ne s'appliquent qu'à des fonctions et non à d'autres fonctionnelles. Nous distinguons le profil d'une fonction en

Deuxième partie

le notant " $\rightarrow$ " de celui de l'opération "o" (qui est une fonctionnelle) en le notant par " $\rightarrow$ ". Le polymorphisme de "o" est limité, "o" ne permet pas de composer des fonctionnelles en FP.

Le profil de " $\_$ " est :

$\_ : \alpha \rightarrow \beta \times \xi \rightarrow \delta \rightarrow \alpha \times \xi \rightarrow \beta \times \delta$

et celui de " $\_-\_;\_$ " est:

$\_-\_;\_ : \alpha \rightarrow \text{bool} \times \alpha \rightarrow \beta \times \alpha \rightarrow \beta \rightarrow \alpha \rightarrow \beta$

où bool désigne le type booléen.

Le profil de "ALL" est:  $\alpha \rightarrow \beta \rightarrow S \rightarrow S$

si  $S$  désigne le type séquence. Mais ce n'est pas assez précis, il est nécessaire de désigner le type des éléments des séquences par une variable de type qui fait du type séquence un type "paramétré" ou "générique" (ce qui est encore un cas particulier de polymorphisme). Le profil de ALL est alors

$ALL: \alpha \rightarrow \beta \rightarrow S(\alpha) \rightarrow S(\beta)$ .

Le profil de "\*" est

$*: \alpha \times \beta \rightarrow \xi \rightarrow \alpha \times S(\beta) \rightarrow S(\xi)$ .

On peut définir ainsi le type des opérations de la FP-algèbre.

##### b) Considérons les symboles des primitives

$PRIM = \{id, ?1, ?2, last, tl, trans, dist1, \dots, +, -, mult, \dots\}$

Deuxième partie

Les primitives sont des fonctions, mais ce sont des "constantes" du point de vue de la FP-algèbre. Ce sont donc des "constantes (fonctionnelles)" à valeur de type fonction. Par exemple "id" engendre une fonction " $a \mapsto a$ ", ce que l'on note:

$\text{id} : \rightarrow a \mapsto a$

On a (en se limitant aux deux sélecteurs ?1 et ?2)

$?1 : \rightarrow a \times \beta \mapsto a$  et

$?2 : \rightarrow a \times \beta \mapsto \beta$ .

Pour les symboles des opérations sur les séquences, le profil " $t1 : \rightarrow S(a) \mapsto S(a)$ " définit  $t1$  comme une fonction partielle. Nous préférons la définir comme une opération totale sur le type des suites non vides que nous notons ici SNV

$t1 : \rightarrow \text{SNV}(a) \mapsto S(a)$

Le type SNV est une "sous-sortie" de  $S$  ( $\text{SNV} \subset S$ ).

On sait décrire le profil de last :  $\rightarrow \text{SNV}(a) \mapsto a$

On peut faire de même pour toutes les constantes de la FP-algèbre. Par exemple

$+: \rightarrow \text{ENT} \times \text{ENT} \mapsto \text{ENT}$  où ENT désigne le type des entiers naturels.

#### c) Cas particulier de la primitive trans

Le profil de trans est

$\text{trans} : \rightarrow S(a) \times S(\beta) \mapsto S(a \times \beta)$

Deuxième partie

mais ceci est insuffisant pour décider de la définition d'un terme car "trans" doit s'appliquer à un couple de séquences de même longueur. "trans" est une opération partielle sur les couples de séquences mais l'expression de la sous-sortie des couples de séquences de même longueur s'exprime par une condition (c'est dans le langage OBJ2 [Fut84] l'expression d'une contrainte de type).

$\text{trans} : \rightarrow SE(a, \beta) \mapsto S(a \times \beta)$  où SE désigne le type couple de séquences de même longueur. On a  $SE \subset S \times S$ .

SE est défini par la contrainte sur "trans":  $\text{trans } o [a, b]$  SI  $\text{length } o a = \text{length } o b$ . "length" est une primitive donnant la longueur d'une séquence et il faut alors inclure les axiomes de length dans la FP-algèbre.

#### d) Cas des fonctions constantes

Nous avons noté CONST l'ensemble des fonctions constantes résultats de l'application de " $\_$ ". On a par exemple

$2^- : \rightarrow a \mapsto \text{ENT}$

On sait ainsi décrire le type d'un élément quelconque de CONST.

Nous avons ainsi fortement typé la FP-algèbre qui devient ainsi une algèbre multi-sortes. On a, de plus, (sur le modèle des spécifications de types proposées pour OBJ [Fut84] et [Gog85]) affecté un ordre partiel entre les types. La FP-algèbre ainsi typée est une algèbre multi-sortes ordonnée.

Deuxième partie

## 1.2. Systèmes de réécriture de règles à prémisses "t est bien formé"

La décision "t est bien formé" peut se faire avec un algorithme convenable de contrôle de type (type checking). Des algorithmes de type-checking ont été prévus pour l'implémentation du langage ML [Mil78]. Une procédure décidant si un terme est "bien formé" a pour donnée la déclaration des types de tous les symboles et doit aussi savoir vérifier si ces déclarations sont consistantes. Elle s'applique à des algèbres dites "multi-sortes" avec en plus des symboles ayant leur type défini avec des variables, qu'on appelle des types "polymorphes".

Notons que, si on ajoute la contrainte de type apportée par les sous-types, au prédicat "t est bien formé", cela est équivalent à "t est défini". On peut considérer "t est bien formé" comme prémisses dans un système de réécriture conditionnel. La vérification de la prémisses "t est bien formé" est faite par une procédure de contrôle de type.

Il est plus judicieux d'utiliser une approche différente qui est réalisée pour le langage OBJ ([Fut84],[Gog85]). Les inclusions de type  $sCs'$  sont traduites par une opération dite de coercion  $c_{s,s'}$  de profil  $s \rightarrow s'$ . On ajoute à l'ensemble des axiomes les axiomes dûs aux coercions qui assurent que l'opération  $c_{s,s'}$  est transitive et que l'inclusion des types est transitive. Quand on obtient, par inclusion des types, une opération, par exemple "+", définie pour les deux types s et s' tels que  $sCs'$ , on ajoute une équation "de morphisme" pour indiquer que le résultat est le même pour s et s'.

$$c_{s,s'}(x) + c_{s,s'}(y) = c_{s,s'}(x+y)$$

Deuxième partie

(+ note l'opération du type s et +' celle du type s'). Ceci permet de traduire l'algèbre typée dans un ensemble d'axiomes classiques. Si A est l'ensemble des équations de l'algèbre multi-sortes ordonnées, on obtient une traduction  $A^*$  de A en ajoutant à A les équations apportées par les coercions. Il y a équivalence entre la classe des algèbres multi-sortes ordonnées validant A et la classe des algèbres classiques validant  $A^*$ . Il faut traduire les opérations (comme +) qui sont "surchargées" par l'inclusion des types par plusieurs opérations dans  $A^*$  (comme + et +'). On choisit l'opération de plus petite coarité telle que l'expression résultat soit bien formée. On peut alors appliquer une procédure de Knuth-Bendix à l'ensemble d'axiomes  $A^*$ .

Si nous utilisons cette approche pour notre FP-algèbre typée, cela revient à supprimer les prémisses "t est bien formé" par la description des types des opérations.

## 2. Systèmes de réécriture conditionnels

Nous considérons maintenant les règles qui ont des prémisses autres que le prédicat "p est bien formé". Il nous reste deux formes de règles

$$\bullet (1) \quad g \Rightarrow d \text{ SI } \text{length } o \ x = \text{length } o \ y \text{ où } x \text{ et } y \in V(g)$$

qui proviennent de la contrainte de type définissant SE type couple de séquences de même longueur.

$$\bullet (2) \quad g \Rightarrow d \text{ SI } p \ o \ [x,a] \text{ où } p, x, a \in V(g)$$

qui sont des règles sur "WH(p,f) o [x,a]".

Deuxième partie

L'approche proposée par [Rem82] traite les prémisses de la forme "précondition" comme celle de la règle (2). Mais les conditions sont trop restrictives: les prémisses doivent pouvoir être évaluées dans le système à la base de la hiérarchie (qui contient les axiomes des booléens) ce qui oblige à limiter les substitutions dans les prémisses par des termes éléments de cette algèbre de base. Or les variables situées dans les prémisses citées ci-dessus doivent pouvoir être substituées par des FP-termes.

L'approche proposée par [Kap84] est directement utilisable pour le cas des règles de la forme (1).

Elle exige:

a) des prémisses de la forme  $u_1 = v_1 \ \& \ u_2 = v_2 \ \& \dots \ \& \ u_n = v_n$ , La règle (1) a des prémisses de cette forme (mais ce n'est pas exactement le cas de la règle (2)).

b) un ordre de simplification qui permet de conclure que  $g > d$  et  $g > m$  et  $g > n$  pour une règle  $g \Rightarrow d$  si  $m = n$ . Le problème ne vient pas des conditions  $g > m$  ou  $g > n$ . Mais il est essentiel de trouver un ordre de simplification pour prouver  $g > d$  or l'ordre que nous utilisons pour prouver la terminaison de nos systèmes de réécriture (voir le chapitre I de la deuxième partie) n'est pas un ordre de simplification et pour pouvoir utiliser l'approche de [Kap84] pour les règles de la forme (1), il faudrait être en mesure d'orienter nos axiomes dans l'ordre qui nous convient (pour la suppression des séquences intermédiaires) avec un ordre de simplification, ce que l'on peut espérer pouvoir faire en utilisant résultats futurs de [Gna85].

Deuxième partie

### 3. Règles de la forme $m = n \mid - g \Rightarrow d$

● Nous considérons les deux règles R1 et R2 que nous avons définies pour des théories particulières au chapitre II de la première partie.

R1:  $PA(g,f) \mid - \text{ALL } g \circ (\text{dist1 } o [x,*f \circ [x,a]]) = *f \circ [g \circ [x,z],a]$

avec  $PA(g,f): g \circ [a,f \circ [b,c]] = f \circ [g \circ [a,b],c]$  pour tout  $a,b,c$ .

R2:  $PB(g,f) \mid - \text{ALL } g \circ (\text{trans } o [*f \circ [x,a],*f \circ [y,b]])$

$= *f \circ [g \circ [x,y], \text{ALL } g \circ (\text{trans } o [a,b])]$

avec  $PB(g,f): g \circ [f \circ [x,y], f \circ [a,b]] = f \circ [g \circ [x,a], g \circ [y,b]]$  pour tout  $x,y,a,b$ .

Les axiomes R1 et R2 sont de la forme

(pour tout  $z$ ,  $P(x,y,z) \mid - g(x,y) = d(x,y)$ )

où  $P(x,y,z)$  est une équation " $m(x,y,z) = n(x,y,z)$ " (Pour plus de clarté, nous avons ici noté les ensembles de variables des termes  $g$ ,  $d$  et  $P$  entre parenthèses à côté du terme).

Il est possible de traiter ces règles avec l'approche de S. Kaplan. En effet, prouver que, pour une substitution  $\sigma$ ,

pour tout  $z$ ,  $m(\sigma(x),\sigma(y),z) = n(\sigma(x),\sigma(y),z)$

équivalent à prouver que

$m(\sigma(x),\sigma(y),z) = n(\sigma(x),\sigma(y),z)$

en considérant  $z$  comme une constante (c'est à dire que  $z$  n'est pas substituable). R1 et R2 peuvent donc être traitées comme de simples règles conditionnelles.

Deuxième partie

● Règle R3

R3:  $PC(g, f, h, s) \vdash g \circ [last \circ (*f \circ [x, iota \circ n]), s \circ n]$   
 $= last \circ (*h \circ [g \circ [x, 1], ALL \ s \circ (iota \circ n)])$

avec  $PC(g, f, h, s): g \circ [f \circ [x, i], s \circ i] = h \circ [g \circ [x, i], s \circ i]$  pour tout  $x, i$ .

La règle R3 est de la forme

$(\text{pour tout } z, P(x, y, z)) \vdash g(x) = g(y)$

où  $P(x, y, z)$  est une équation  $m(x, z) = n(y, z)$ .

Il s'agit, si la règle est orientée de gauche à droite, de trouver une substitution  $\phi$  telle que " $m(\sigma(x), z) = n(\phi(y), z)$ " pour une substitution  $\sigma$  de domaine  $x$ . Autrement dit de résoudre l'équation " $m(\sigma(x), z) = n(y, z)$ " pour la variable  $y$ . Cette résolution d'équation permet de trouver la substitution  $\phi(y)$  et donc d'avoir des règles de réécriture qui ont dans leur membre gauche des variables qui ne figurent pas dans le membre droit. Les systèmes de réécriture conditionnels n'acceptent pas encore d'intégrer des résolutions d'équations au niveau des prémisses. C'est un problème ouvert intéressant à résoudre (il se pose aussi en programmation logique).

### Conclusion

Nous n'avons donné ici que quelques idées. Une grande partie du travail reste à faire.

Les prémisses de la forme "DEF o p" peuvent se ramener à un contrôle de type. Mais il faut avoir une procédure de "type checking" pour une algèbre à types "ordonnés" incluant le polymorphisme. Notons alors que, ce faisant, nous introduisons les notions de type qui n'existent pas en FP. Nous

voulons utiliser une approche semblable à celle proposée pour le contrôle de type dans le langage ML [Mil78] plutôt que celle qui est proposée pour le contrôle des types en FP [Gut81].

Nous pensons d'ailleurs qu'il est préférable de transformer une spécification de l'algèbre FP typée dans une algèbre "non ambiguë" qui permette de supprimer les prémisses "DEF o p" qui s'apparentent à un contrôle de type comme cela est prévu pour le langage OBJ [Fut84].

Pour les systèmes de réécriture conditionnels, nous nous proposons d'essayer d'utiliser les résultats de [Kap84], ce qui nous permet d'inclure les règles R1 et R2 du chapitre II de la première partie.

## CONCLUSION

Nous allons résumer la ligne directrice de ce travail et rappeler quels sont les problèmes ouverts et les prolongements que nous pouvons envisager.

Quand on effectue une transformation de programmes, c'est toujours dans un certain but, par exemple supprimer la récursivité. Le nôtre est d'éliminer, dans les programmes itératifs, les séquences intermédiaires inutiles (Ces séquences intermédiaires sont engendrées par les calculs d'une machine de type Von Neumann).

Nous considérons que la transformation d'un programme est le résultat de la réécriture d'une expression fonctionnelle codant ce programme.

Définir le système de transformation revient alors à

1) Définir une algèbre  $F$  des fonctions qui codent une classe de programmes (dans notre cas, les programmes itératifs, sachant que nous nous restreignons à l'étude de ceux qui se terminent). Nous avons résolu ce problème simplement en codant les programmes dans le langage FP et nous travaillons dans le cadre d'une FP-algèbre.

2) Définir un ensemble  $A$  d'équations validées par  $F$  tel que la congruence syntaxique engendrée par la  $A$ -égalité donne des classes d'équivalences qui contiennent chacune un élément qui soit le résultat voulu comme transformé de tous les termes de la classe.

Dans notre cas, cet élément doit engendrer le nombre minimum de séquences intermédiaires. Pour cela, il suffit d'inclure dans l'ensemble d'axiomes  $A$  tous les axiomes qui concernent les compositions des fonctions

à arguments ou/et à résultats séquences.

3) Trouver un système de réécriture  $R$  qui satisfasse aux deux critères suivants:

a)  $R$  doit être "équivalent" à l'ensemble d'axiomes  $A$ , c'est-à-dire que deux expressions sont  $A$ -égales si elles ont la même forme normale. Pour cela, le système de réécriture  $R$  doit être convergent.

b) La forme normale d'un terme  $t$  doit engendrer un minimum de séquences intermédiaires.

Le critère a est satisfait si  $R$  est le résultat d'une procédure de Knuth-Bendix appliquée à l'ensemble d'axiomes  $A$ . Le critère b est satisfait si l'orientation des règles coïncide avec l'ordre qui permet d'éliminer des séquences intermédiaires.

Le problème devient alors plus difficile.

Sur le plan pratique, nous sommes limités par les restrictions imposées par les implémentations de la procédure de Knuth-Bendix. Par exemple, nous utilisons une version simple qui n'accepte pas la réécriture modulo un ensemble d'équations ce qui nous interdit d'inclure les axiomes permutatifs.

Mais la difficulté fondamentale est que nous voulons une orientation précise des équations en des règles de réécriture qui éliminent des séquences intermédiaires et que cette orientation ne coïncide pas avec les ordres implémentés dans la procédure pour orienter les règles. Aussi quand nous obtenons un système de réécriture, nous ne sommes assurés que de sa

confluence locale et nous devons prouver que le système est noethérien. Nous avons pu constater que nous ne savions pas prouver cette propriété d'une manière satisfaisante.

Nous nous heurtons aussi à la principale difficulté d'utilisation d'une procédure de Knuth-Bendix. La procédure n'est pas toujours capable d'engendrer en un nombre fini d'étapes un système ayant la propriété de confluence locale. Nous avons pu constater les divergences de la procédure provoquées par certains de nos axiomes. Théoriquement, la procédure de Knuth-Bendix engendre, dans ce cas, un système infini de règles qui a la propriété de confluence locale. Nous avons envisagé la possibilité d'utiliser un système de méta-réécriture qui soit équivalent à un système de réécriture contenant des familles infinies de règles.

Sur le plan théorique, nous rencontrons d'autres difficultés. Notre FP-algèbre valide des axiomes conditionnels et les systèmes de réécriture conditionnels sont d'utilisation limitée par le problème de la décidabilité de la réécriture conditionnelle. Nous pouvons cependant mettre à part les axiomes qui ne concernent que des fonctions d'un certain profil. Si nous considérons une algèbre FP typée, ces conditions tombent. On est ramené au problème de la traduction de cette algèbre typée dans une algèbre classique qui tienne compte des axiomes ajoutés par les coercions de type. Il nous semble que les quelques axiomes conditionnels qui restent peuvent être traités par l'approche développée par S.Kaplan.

Notre travail montre que cette démarche n'est pas utopique.

Pour construire un système de réécriture capable d'effectuer les transformations de nos programmes codés en FP, nous avons utilisé la

Deuxième partie

procédure de Knuth-Bendix implémentée dans la version 2.1 du système REVE.

Nous avons dû orienter les règles avec l'option "manuel"; la procédure oriente alors les règles selon l'ordre indiqué par l'utilisateur. Nous utilisons diverses stratégies d'orientation des règles et de "rejet" des paires critiques non valides pour tenir compte des problèmes de définition (qui sont des problèmes de type) des expressions.

Nous avons ainsi construit des systèmes de réécritures par inclusions successives d'ensembles d'axiomes qui concernent, les fonctionnelles de base, les primitives sur les séquences, puis les fonctionnelles "ALL", "\*" et enfin "WH". Nous obtenons des systèmes croissants (pour l'ordre d'inclusion) de réécriture convergents. Les axiomes que nous n'avons pu inclure provoquent des divergences de la procédure.

Le système de réécriture obtenu normalise les expressions codant les programmes en supprimant les séquences intermédiaires et donnent les mêmes résultats qu'un système de transformation qui fusionne et compose les itérations.

Il est aussi intéressant de laisser diverger la procédure et de pouvoir alors engendrer des règles qui améliorent les résultats obtenus.

Il est évident que, pour obtenir ces seuls résultats, il aurait été plus simple de construire directement un système classique de transformation. Mais un aspect intéressant de notre travail est de montrer la possibilité d'utiliser les techniques de réécriture à partir de l'algèbre de fonctions pour la transformation des programmes.

Les prolongements que nous envisageons sont liés, d'une part, au

Deuxième partie

développement des techniques de réécriture.

Nous envisageons de suivre l'évolution du système REVE et d'utiliser les nouvelles possibilités qui seront offertes par la version REVE 3 qui accepte la réécriture modulo une théorie équationnelle.

Nous entrevoyons des possibilités nouvelles pour faciliter les preuves de terminalison, par exemple en utilisant un logiciel intégré dans la version REVE 2.2 qui accepte d'orienter les règles avec une interprétation polynomiale donnée.

Nous pouvons envisager d'améliorer nos résultats actuels avec une implémentation de la méta-réécriture.

Nous devons aussi pouvoir essayer prochainement l'approche de S. Kaplan (dont une implémentation est en cours) pour les systèmes de réécriture conditionnels.

Mais, d'autre part, nous sommes aussi très intéressés par les règles sur l'itérateur "\*" liées aux théories particulières (sur les entiers, les tableaux,...) dont nous avons donné un aperçu au chapitre II de la première partie. Ces règles correspondent à des théorèmes puissants sur l'équivalence des programmes itératifs et mériteraient une étude plus approfondie (travail qui est envisagé par ([Kie81], [Jay83]) pour la transformation des définitions FP récursives).

Par ailleurs, il serait intéressant d'essayer d'obtenir un système de réécriture capable de transformer les programmes (itératifs) pour augmenter les possibilités de calculs "en parallèle", c'est-à-dire d'entreprendre une étude similaire pour les programmes parallèles.

## Plan des annexes

### ANNEXE I: ENSEMBLE D'AXIOMES DE LA FP-ALGEBRE

|                                                       |   |
|-------------------------------------------------------|---|
| 1. Axiomes de base .....                              | 4 |
| 2. Axiomes sur " $\rightarrow$ " .....                | 4 |
| 3. Axiomes sur "ALL" .....                            | 5 |
| 4. Axiomes sur "*" .....                              | 6 |
| 4.1. Famille des fusions et compositions .....        | 6 |
| 4.2. Axiomes avec les primitives .....                | 6 |
| 5. Axiomes sur "WH" .....                             | 7 |
| 5.1. Compositions .....                               | 7 |
| 5.2. Cas particulier de " $p \circ ?2$ " .....        | 8 |
| 5.3. Axiomes avec les primitives .....                | 8 |
| 6. Axiomes sur les primitives sur les séquences ..... | 9 |
| 7. Axiomes sur les autres primitives                  |   |

### ANNEXE II : LES SYSTEMES DE REECRITURE CONVERGENTS

|                                                                            |    |
|----------------------------------------------------------------------------|----|
| 1. Avec les axiomes E1 et E2 .....                                         | 13 |
| 2. Avec les axiomes de base .....                                          | 13 |
| 3. Avec les axiomes sur les primitives sur les séquences, .....            | 14 |
| 4. Avec les axiomes de "ALL" .....                                         | 17 |
| 5. Avec l'axiome supplémentaire ID: $[?1, ?2] \Rightarrow \text{id}$ ..... | 19 |
| 6. Avec les axiomes sur "*", .....                                         | 24 |
| 7. Avec les axiomes de "WH", .....                                         | 25 |

8. Avec les axiomes de " $\_ \rightarrow \_ ; \_$ ",

### ANNEXE III : LES META-SYSTEMES DE REECRITURE CONVERGENTS

|                                                                 |    |
|-----------------------------------------------------------------|----|
| 1. Avec les axiomes E1 et E2 .....                              | 30 |
| 2. Avec les axiomes de base .....                               | 31 |
| 3. Avec les axiomes sur les primitives sur les séquences, ..... | 31 |
| 4. Avec les axiomes de "ALL" .....                              | 34 |
| 5. Avec l'axiome supplémentaire ID .....                        | 36 |
| 6. Avec les axiomes sur " $\ast$ ", .....                       | 40 |
| 7. Avec les axiomes de " $\_ \rightarrow \_ ; \_$ ",            |    |

### ANNEXE 4: DES EXEMPLES

|                                                      |    |
|------------------------------------------------------|----|
| 1. Exemple 1: calcul de $\$2$ sup $n\$$ .....        | 43 |
| 2. Exemple 2: calcul de $\$x$ sup $n\$$ .....        | 43 |
| 3. Exemple 3: calcul de $\$2$ sup $n\$/n!$ .....     | 44 |
| 4. Exemple 4: .....                                  | 46 |
| 5. Exemple 5: .....                                  | 47 |
| 6. Exemple 6: des calculs de moyennes .....          | 48 |
| 7. Exemple 7: moyenne simple de 10 entiers .....     | 50 |
| 8. Exemples 8 et 9: La réduction des conditionnelles |    |

### ANNEXE 5: PREUVE DE TERMINAISON

|                                                 |    |
|-------------------------------------------------|----|
| 1. Systèmes S-BASE et S-BASE' .....             | 51 |
| 1.1. Règles E1 et E2 .....                      | 51 |
| 1.2. Règles restantes des systèmes S-BASE ..... | 52 |
| 2. Systèmes S-ALL .....                         | 52 |
| 2.1. Règles ALLALL et ALLALL' .....             | 52 |

|                                                         |    |
|---------------------------------------------------------|----|
| 2.2. Règle d'associativité de "append" .....            | 53 |
| 2.3. Règles TLALL et TLALL' .....                       | 54 |
| 2.4. Règle TALL .....                                   | 56 |
| 2.5. Règle entre "dist1" et "append" .....              | 57 |
| 2.6. Autres règles des systèmes S-ALL .....             | 58 |
| 3. Systèmes de réécriture sur les primitives .....      | 61 |
| 4. Systèmes de réécriture S- $\ast\ast$ et S-WHWH ..... | 64 |
| 4.1. Interprétation de " $\ast f$ " .....               | 64 |
| 4.2. Interprétation de " $WH(p, f)$ " .....             | 67 |
| 5. Systèmes de réécriture S- $\ast$ et S-WH             |    |
| BIBLIOGRAPHIE .....                                     | 70 |

# ANNEXE I: ENSEMBLE D'AXIOMES DE LA FP-ALGEBRE

Nous considérons l'ensemble d'axiomes A suivant:

## 1. Axiomes de base

$$E1: (f \circ g) \circ h = f \circ (g \circ h)$$

$$E2: [f1, f2, \dots, fn] \circ h = [f1 \circ h, f2 \circ h, \dots, fn \circ h]$$

$$E3: DEF \circ y :: x^- \circ y = x^-$$

$$P1: id \circ f = f$$

$$P2: f \circ id = f$$

$$P3: DEF \circ [f1, f2, \dots, fn] :: ?i \circ [f1, f2, \dots, fi, \dots, fn] = fi$$

## 2. Axiomes sur "→;\_"

$$COND1: (p \rightarrow f; g) \circ h = (p \circ h) \rightarrow (f \circ h); (g \circ h)$$

$$COND2: h \circ (p \rightarrow f; g) = p \rightarrow (h \circ f); (h \circ g)$$

$$COND3: p \rightarrow (p \rightarrow f; g); h = p \rightarrow f; h$$

$$COND4: p \rightarrow f; (p \rightarrow g; h) = p \rightarrow f; h$$

$$COND5: [h1, \dots, hi-1, (p \rightarrow f; g), hi+1, \dots, hn] \\ = p \rightarrow [h1, \dots, hi-1, f, hi+1, \dots, hn]; [h1, \dots, hi-1, g, hi+1, \dots, hn]$$

$$C: p \rightarrow (q \rightarrow f; g); (q \rightarrow x; y) = q \rightarrow (p \rightarrow f; x); (p \rightarrow g; y)$$

$$T: (p \rightarrow x; y) \rightarrow f; g = p \rightarrow (x \rightarrow f; g); (y \rightarrow f; g)$$

$$TRUE: T^- \rightarrow f; g = f$$

$$FALSE: F^- \rightarrow f; g = g$$

$$PPX: p \rightarrow p; x = p \rightarrow T^-; x$$

$$PXP: p \rightarrow x; p = p \rightarrow x; F^-$$

## 3. Axiomes sur "ALL"

$$ALLALL: ALL f \circ ALL g = ALL (f \circ g)$$

$$FALL: trans \circ [ALL f1, ALL f2, \dots, ALL fn] = ALL[f1, f2, \dots, fn]$$

$$HALL: DEF \circ ALL f :: hd \circ ALL f = f \circ hd$$

$$LALL: DEF \circ ALL f :: last \circ ALL f = f \circ last$$

$$TLALL: DEF \circ ALL f :: tl \circ ALL f = ALL f \circ tl$$

$$ALLID: ALL id = id$$

$$ALLVID: ALL f \circ \langle \rangle^- = \langle \rangle^-$$

$$ALLCONS: cons \circ [f \circ x, ALL f \circ a] = ALL f \circ (cons \circ [x, a])$$

$$ALLAPP: append \circ [ALL f \circ a, ALL f \circ h] = ALL f \circ (append \circ [a, h])$$

$$ALL?iTRANS: DEF \circ (trans \circ [a1, a2, \dots, an]) ::$$

$$ALL?i \circ (trans \circ [a1, a2, \dots, ai, \dots, an]) = ai$$

$$ALLDISTL: ALL[f \circ ?1, g \circ ?2] \circ (distl \circ [x, a]) = distl \circ [f \circ x, ALL g \circ a]$$

#### 4. Axiomes sur "\*"

\*ALL:  $*f \circ [x, ALL \ g \circ h] = *(f \circ [?1, g \circ ?2]) \circ [x, h]$

##### 4.1. Famille des fusions et compositions

F\*:  $trans \circ [*f_1 \circ [x_1, a], *f_2 \circ [x_2, a], \dots, *f_n \circ [x_n, a]]$   
 $= [*f_1 \circ [?1o?1, ?2], f_2 \circ [?2o?1, ?2], \dots, f_n \circ [?no?1, ?2]]$   
 $\circ [[x_1, x_2, \dots, x_n], a]$

F:  $trans \circ [ \dots,$   
 $\quad *f_i \circ [x_i, a], \dots, ALL \ f_j \circ (*f_k \circ [x_k, a]),$   
 $\quad \dots, *f_l \circ [x_l, t_1 \circ (*f_m \circ [x_m, a])],$   
 $\quad \dots, *f_k \circ [x_k, a], \dots, *f_m \circ [x_m, a], \dots$   
 $\quad \dots]$   
 $= [* \dots, f_i \circ [?1o?1, ?2],$   
 $\quad \dots, f_j \circ (f_k \circ [?ko?1, ?2]),$   
 $\quad \dots, f_l \circ [?1o?1, f_m \circ [?mo?1, ?2]],$   
 $\quad \dots, f_k \circ [?ko?1, ?2], \dots, f_l \circ [?1o?1, ?2], \dots]$   
 $\circ [ \dots, x_i, \dots, f_j \circ x_k, \dots, x_1, \dots, x_k, \dots, x_m, \dots], a]$

##### 4.2. Axiomes avec les primitives

H\*:  $DEF \ n \ (*f \ n \ [x, a]) :: hd \ n \ (*f \ n \ [x, a]) = x$

TL\*:  $DEF \ o \ (t_1 \ o \ a) :: t_1 \ o \ (*f \ o \ [x, a]) = *f \ o \ [f \ o \ [x, hd \ o \ a], t_1 \ o \ a]$

\*VID:  $*f \ o \ [x, <>] = cons \ o \ [x, <>]$

\*CONS:  $cons \ o \ [z, *f \ o \ [f \ o \ [z, x], a]] = *f \ o \ [z, cons \ o \ [x, a]]$

TL\*?1:  $t_1 \ o \ *?1 = ALL?1 \ o \ dist_1$

\*?2:  $*?2 = cons$

\*APP:  $append \ o \ [*f \ o \ [x, a], t_1 \ o \ (*f \ o \ [last \ o \ (*f \ o \ [x, a]), b])]$   
 $= *f \ o \ [x, append \ o \ [a, b]]$

#### 5. Axiomes sur "WH"

##### 5.1. Compositions

WHALL:  $WH(p, f) \circ [x, ALL \ g \circ h]$   
 $= WH(p \circ [?1, g \circ ?2], f \circ [?1, g \circ ?2]) \circ [x, a]$   
 ALLWH:  $trans \circ [ALL \ g \circ (WH(p, f) \circ [x, h]), WH(p, f) \circ [x, h]]$   
 $= WH(p \circ [?2o?1, ?2], [g \circ f, f] \circ [?2o?1, ?2]) \circ [[g \circ x, x], h]$   
 ALLWHS:  $trans \circ [WH(p, f) \circ [x, h], ALL \ g \circ (WH(p, f) \circ [x, h])]$   
 $= WH(p \circ [?1o?1, ?2], [f, g \circ f] \circ [?1o?1, ?2]) \circ [[x, g \circ x], h]$   
 \*WH:  $trans \circ [*f \ o \ [x, t_1 \ o \ (WH(p, g) \ o \ [y, h])], WH(p, g) \ o \ [y, h]]$   
 $= WH(p \circ [?2o?1, ?2], [f, ?2] \ o \ [?1o?1, g \ o \ [?2o?1, ?2]]) \ o \ [[x, y], h]$   
 \*WHS:  $trans \circ [WH(p, g) \ o \ [y, h], *f \ o \ [x, t_1 \ o \ (WH(p, f) \ o \ [y, h])]]$   
 $= WH(p \circ [?1o?1, ?2], [?2, f] \ o \ [?2o?1, g \ o \ [?1o?1, ?2]]) \ o \ [[y, x], h]$

WH\* et WH\*S:  $WH(p,f) \circ [x,t1 \circ (*g \circ [y,h])]$   
 $= ALL?1$   
 $\circ ( WH(p \circ [?1o?1,g \circ [?2o?1,?2]], [f,?2] \circ [?1o?1,g \circ [?2o?1,?2]])$   
 $\circ [[x,y],h])$   
 $= ALL?2$   
 $\circ ( WH(p \circ [?2o?1,g \circ [?1o?1,?2]], [?2,f] \circ [?2o?1,g \circ [?1o?1,?2]])$   
 $\circ [[y,x],h] )$   
 WHWH et WHWH\*:  $WH(p,f) \circ [x,t1 \circ (WH(q,g) \circ [y,h])]$   
 $= ALL?1 \circ WH((q \circ [?2o?1,?2])$   
 $\rightarrow ((p \circ [?1o?1,g \circ [?2o?1,?2]]) \rightarrow T^-; F^-); F^-$   
 $, [f,?2] \circ [?1o?1,g \circ [?2o?1,?2]]) \circ [[x,y],h]$   
 $= ALL?2 \circ WH((q \circ [?1o?1,?2])$   
 $\rightarrow ((p \circ [?2o?1,g \circ [?1o?1,?2]]) \rightarrow T^-; F^-); F^-$   
 $, [?2,f] \circ [?2o?1,g \circ [?1o?1,?2]]) \circ [[y,x],h]$

### 5.2. Cas particulier de "p o ?2"

WHPART:  $DEF \circ (WH(p \circ ?2,?2) \circ [y,a]) ::$   
 $*f \circ [x,t1 \circ (WH(p \circ ?2,?2) \circ [y,a])] = WH(p \circ ?2,f) \circ [x,a]$

### 5.3. Axiomes avec les primitives

HWH:  $DEF \circ (WH(p,f) \circ [x,a]) :: hd \circ (WH(p,f) \circ [x,a]) = x$   
 TLWH:  $(p \circ [x,hd \circ a]) :: t1 \circ (WH(p,f) \circ [x,a])$   
 $= WH(p,f) \circ [f \circ [x,hd \circ a], t1 \circ a]$   
 WHVID:  $WH(p,f) \circ [x,<>^-] = cons \circ [x,<>^-]$

WHCONS:  $p \circ [z,x] :: cons \circ [z,WH(p,f) \circ [f \circ [z,x],a]]$   
 $= WH(p,f) \circ [z,cons \circ [x,a]]$

### 6. Axiomes sur les primitives sur les séquences

#### Axiome "last" et "t1"

LTL:  $DEF \circ (last \circ t1) :: last \circ t1 = last$

#### Sur "trans"

HTRANS:  $DEF \circ (trans \circ [a1,...,an]) ::$   
 $hd \circ (trans \circ [a1,...,an]) = [hd \circ a1,...,hd \circ an]$

LTRANS:  $DEF \circ (trans \circ [a1,...,an]) ::$   
 $last \circ (trans \circ [a1,...,an]) = [last \circ a1,...,last \circ an]$

TLTRANS:  $t1 \circ (trans \circ [a1,...,an]) = trans \circ [t1 \circ a1,...,t1 \circ an]$

TCONS:  $trans \circ [cons \circ [x1,a1],...,cons \circ [xn,an]]$   
 $= cons \circ [[x1,...,xn], trans \circ [a1,...,an]]$

TVID:  $TRANS \circ [<>^-,...,<>^-] = <>^-$

#### Sur "dist1"

HDISTL:  $hd \circ (dist1 \circ [x,a]) = [x,hd \circ a]$

LDISTL:  $last \circ (dist1 \circ [x,a]) = [x,last \circ a]$

TLDISTL:  $t1 \circ (dist1 \circ [x,a]) = dist1 \circ [x,t1 \circ a]$

TDISTL:  $trans \circ [ALL?1 \circ (dist1 \circ [x,a]),a] = dist1 \circ [x,a]$

DCONS:  $dist1 \circ [x,cons \circ [y,a]] = cons \circ [[x,y],dist1 \circ [x,a]]$

DVID: DEF o x :: distl o [x,<>~] = <>~

Sur "cons"

HCONS: DEF o (cons o [x,a]) :: hd o (cons o [x,a]) = x

TLCONS: DEF o (cons o [x,a]) :: tl o (cons o [x,a]) = a

LCONS: (DEF o x) & (DEF o (last o a)) ::

last o (cons o [x,a]) = last o a

CONS: DEF o (tl o a) :: cons o [hd o a,tl o a] = a

LCVID: last o (cons o [x,<>~]) = x

Sur "append"

HAPP: (DEF o (append o [a,b])) & (DEF o (hd o a)) ::

hd o (append o [a,b]) = hd o a

TLAPP: DEF o (tl o a) :: tl o (append o [a,b]) = append o [tl o a,b]

LAPP: (DEF o (append o [a,b])) & (DEF o (last o b)) ::

last o (append o [a,b]) = last o b

APPCONS: append o [cons o [x,a],b] = cons o [x,append o [a,b]]

APPVID: DEF o (append o [<>~,a]) :: append o [<>~,a] = a

APPVIDS: DEF o (append o [a,<>~]) :: append o [a,<>~] = a

APPASS: append o [append o [a,b],c] = append o [a,append o [b,c]]

APPDISTL: distl o [x,append o [a,b]]

= append o [distl o [x,a],distl o [x,b]]

APPTRANS: DEF o [trans o [a1,a2,...,an],trans o [b1,b2,...,bn]] ::

append o [trans o [a1,a2,...,an],trans o [b1,b2,...,bn]]

= trans o [append o [a1,b1],append o [a2,b2],...,append o [an,bn]]

Sur "iota"

HIOTA: DEF o iota :: hd o iota = 1~

LIOTA: DEF o iota :: last o iota = id

ALL\*IOTA: ALL(last o (\*f o [g o x~,iota])) o iota

= tl o (\*f o [g o x~,iota])

TL\*IOTA: tl o (\*(+ o [?1,1~]) o [0~,iota]) = iota

APPIOTA: ( (DEF o (- o [n,p])) & (DEF o (iota o p)) ) ::

append o [iota o p,

tl o (\*(+ o [?1,1~]) o [p,iota o (+ o [1~, - o [n,p]])]]

= iota o n

ALL\*IOTA ALL(+ o [1~,id]) o iota = tl o (iota o (+ o [1~,id]))

## 7. Axiomes sur les autres primitives

Nous citons ici uniquement les axiomes utilisés pour les exemples présentés au cours de cette thèse.

### Axiomes de la structure de groupe de "+"

PLUSO: DEF o (+ o [x,0~]) :: + o [x,0~] = x

PLUSASS: + o [a,+ o [b,c]] = + o [+ o [a,b],c]

PLUSCOM: + o [a,b] = + o [b,a]

MOINS:  $+ \circ [a, - \circ [0^-, b]] = - \circ [a, b]$

IMOINS:  $\text{DEF } \circ (\sim \circ [a, a]) :: - \circ [a, a] = 0^-$

Axiomes de la structure de groupe de "mult" dans les reels

MULT1:  $\text{DEF } \circ (\text{mult } \circ [x, 1^-]) :: \text{mult } \circ [x, 1^-] = x$

MULTASS:  $\text{mult } \circ [a, \text{mult } \circ [b, c]] = \text{mult } \circ [\text{mult } \circ [a, b], c]$

MULTCOM:  $\text{mult } \circ [a, b] = \text{mult } \circ [b, a]$

DIV/:  $\text{mult } \circ [a, / \circ [1^-, b]] = / \circ [a, b]$

IDIV/:  $\text{DEF } \circ (/ \circ [a, a]) :: / \circ [a, a] = 1^-$

"0" élément absorbant de "mult"

MULTO:  $\text{mult } \circ [0^-, a] = 0^-$

## ANNEXE II : LES SYSTEMES DE REECRITURE CONVERGENTS

En utilisant REVE, nous avons isolé les systèmes de réécriture convergents suivants:

### 1. Avec les axiomes E1 et E2

```

+++++
+ S-E1E2 a(LR)>[_] +
+ ++++++ +
+ + + + + + +
+ E1: (f o g) o h => f o (g o h) +
+ + + + + + +
+ E2: [f,g] o h => [f o h,g o h] +
+ + + + + + +
+++++

```

```

+++++
+ S-REV-E1E2 [_]>o(RL) +
+ ++++++ +
+ + + + + + +
+ E1: f o (g o h) => (f o g) o h +
+ + + + + + +
+ E2: [f o h,g o h] => [f,g] o h +
+ + + + + + +
+++++

```

### 2. Avec les axiomes de base

```

+++++
+ S-BASE o(LR)>[] +
+ ***** +
+ S-E1E2 et +
+ + +
+ ***** +
+ E3: x~ o y => x~ SI (DEF o y) +
+ + +
+ P1: id o f => f +
+ + +
+ P2: f o id => f +
+ + +
+ P3_1: ?1 o [f,g] => f SI (DEF o [f,g]) +
+ + +
+ P3_2: ?2 o [f,g] => g SI (DEF o [f,g]) +
+ + +
+++++

```

### 3. Avec les axiomes sur les primitives sur les séquences,

on introduit un nouveau symbole "L" défini par

$L(x) = \text{cons } o [x, <>^-]$

qui supprime le symbole "cons".

```

+++++
+ o(LR)>[], id, -, 1, L +
+ {trans, distl}> cons > append +
+ tl>{trans, cons, distl, append} +
+ S-PRIMITIVE +
+ ***** +
+ S-BASE et +
+ + +
+ ***** +
+ LTL: last o tl => last SI (DEF o (last o tl)) +
+ + +
+ LTL': last o (tl o h) => last o h SI (DEF o (last o (tl o h))) +
+ + +
+ LIST: L(x) o y => L(x o y) +
+ + +
+ HLIST: hd o L(x) => x +
+ + +
+ LCVID: last o L(x) => x +
+ + +
+ TLLIST: tl o L(x) => <>^- SI (DEF o x) +
+ + +
+ HTRANS: hd o (trans o [a,b]) => [hd o a, hd o b] +
+ + +
+ SI (DEF o (trans o [a,b])) +
+ + +
+ LTRANS: last o (trans o [a,b]) => [last o a, last o b] +
+ + +
+ SI (DEF o (trans o [a,b])) +
+ + +
+ TLTRANS: tl o (trans o [a,b]) => trans o [tl o a, tl o b] +
+ + +
+ TLIST: trans o [L(x), L(y)] => L([x,y]) +
+ + +
+ HDISTL: hd o (distl o [x,a]) => [x, hd o a] +
+ + +
+ LDISTL: last o (distl o [x,a]) => [x, last o a] +
+ + +
+ TLDISTL: tl o (distl o [x,a]) => distl o [x, tl o a] +
+ + +
+ DLIST: distl o [x, L(y)] => L([x,y]) +
+ + +
+ TVID: trans o [<>^-, <>^-] => <>^- +
+ + +
+ DVID: distl o [x, <>^-] => <>^- SI (DEF o x) +
+ + +
+ HIOTA: hd o iota => 1^- SI (DEF o iota) +
+ + +
+ HIOTA': hd o (iota o n) => 1^- SI (DEF o (iota o n)) +
+ + +
+ LIOTA: last o iota => id SI (DEF o iota) +
+ + +

```

```

+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))
+
+ HAPP: hd o (append o [a,b]) => hd o a
+
+ SI ((DEF o (hd o a)) & (DEF o (append o [a,b])))
+
+ LAPP: last o (append o [a,b]) => last o b
+
+ SI ((DEF o (last o b)) & (DEF o (append o [a,b])))
+
+ TLAPP: tl o (append o [a,b]) => append o [tl o a,b] SI (DEF o (tl o a))
+
+ DEFCONS: cons o [x,a] => append o [L(x),a]
+
+ APPASS: append o [append o [a,b],c] => append o [a,append o [b,c]]
+
+ APPDISTL: distl o [x,append o [a,b]]
+ => append o [distl o [x,a],distl o [x,b]]
+
+ APPTRANS: trans o [append o [a,b],append o [c,d]]
+
+ => append o [trans o [a,c],trans o [b,d]]
+
+ SI (DEF o [trans o [a,c],trans o [b,d]])
+
+ APPVID: append o [a,<>] => a SI (DEF o (append o [a,<>]))
+
+ APPVIDS: append o [<>,a] => a SI (DEF o (append o [<>,a]))
+
+*****

```

## REMARQUE

Le système S-PRIMITIVE est localement-confluent sans la règle APPTRANS. Avec la règle APPTRANS, la procédure REVE se termine si on rejette les paires critiques issues des superpositions avec la règle APPASS que REVE réduit en des équations qui ne sont pas valides car REVE ne tient pas compte de la précondition de la règle APPTRANS (voir le chapitre III de la première partie).

Les paires critiques entre les deux règles APPTRANS et APPASS engendrent

une famille infinie de règles qui peuvent se schématiser par la méta-règle suivante (voir le chapitre II de la deuxième partie):

```

trans o [append o [A,B],append o [C,D]]
=> append o [trans o [A,C],trans o [B,D]]
SI (DEF o [trans o [A,C],trans o [B,D]]).

```

Le domaine des méta-variables A, B, C et D, est l'ensemble des termes en forme normale pour la règle APPASS.

Si l'on considère S-PRIMITIVE comme un méta-système de réécriture, où la règle APPTRANS est remplacée par la méta-règle APPTRANS, S-PRIMITIVE est convergent.

## 4. Avec les axiomes de "ALL"

```

+++++
+ orientation manuelle: S-ALL
+ +++++
+ S-PRIMITIVE et
+ +++++
+ ALLALL: ALL f o ALL g => ALL(f o g)
+
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h
+
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f)
+
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h))
+
+ LALL: last o ALL f => f o last SI (DEF o ALL f)
+
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (last o h))
+
+ TLALL: tl o ALL f => ALL f o tl SI (DEF o ALL f)
+
+ TLALL': tl o (ALL f o h) => ALL f o (tl o h)
+
+ SI (DEF o (ALL f o h))
+
+ ALLID: ALL id => id
+
+ ALLIST: ALL f o L(x) => L(f o x)
+
+ ALLAPP: ALL f o (append o [a,b]) => append o [ALL f o a, ALL f o b]
+
+ ALLVID: ALL f o <>~ => <>~
+
+ FALL: trans o [ALL f, ALL g] => ALL[f,g]
+
+ FALL': trans o [ALL f o h, ALL g o h] => ALL[f,g] o h
+
+ FALLg: trans o [ALL f, id] => ALL[f, id]
+
+ FALLd: trans o [id, ALL g] => ALL[id, g]
+
+ FALL'g: trans o [ALL f o h, h] => ALL[f, id] o h
+
+ FALL'd: trans o [h, ALL f o h] => ALL[id, f] o h
+
+ TALL: trans o [a, a] => ALL[id, id] o a
+
+ TDISTL: trans o [ALL ?l o (distl o [x, a]), a] => distl o [x, a]
+
+++++

```

## REMARQUE

Pour obtenir le système S-\*\*, nous utilisons un sous-ensemble de règles de S-ALL qui forme encore un système de réécriture convergent et que nous appelons S-ALL1. On a

$$S-ALL1 = S-ALL - (\{LTRANS, TLTRANS\} \cup$$

$\{ALLID, ALLIST, ALLAPP, ALLVID, FALLg, FALLd, FALLg', FALLd', TALL, TDISTL\})$

(LTRANS et TLTRANS sont des règles éléments de S-PRIMITIVE).

5. Avec l'axiome supplémentaire ID:  $[?1, ?2] \Rightarrow id$

qu'on ajoute aux axiomes de base

```

+++++
+ S-BASE'
+ +++++
+ S-BASE et
+ +++++
+ ID: [?1, ?2] => id SI (DEF o [?1, ?2])
+
+ ID': [?1 o f, ?2 o f] => f SI (DEF o [?1 o f, ?2 o f])
+
+++++

```

qu'on ajoute aux axiomes sur les primitives sur les séquences

## REMARQUE

L'utilisation de REVE nous oblige (voir le chapitre III de la première partie) à traduire

- append par A(?1,?2),  
 - trans par T(?1,?2),  
 - distl par D(?1,?2) et  
 - cons par A(L(?1),?2).

```

+++++
+ o>{[_],id,1,-,L,T,D,A}
+ t1>{T,D,A}, T>{L,[_],A}, D>{L,[_],A}
+ S-PRIMITIVE'
+ *****
+ S-BASE'
+
+ *****
+ LTL: last o t1 => last SI (DEF o (last o t1))
+
+ LTL': last o (t1 o h) => last o h SI (DEF o (last o (t1 o h)))
+
+ LIST: L(x) o y => L(x o y)
+
+ HLIST: hd o L(x) => x
+
+ LLIST: last o L(x) => x
+
+ TLLIST: t1 o L(x) => <>~ SI (DEF o x)
+
+ TRANS: T(a,b) o h => T(a o h,b o h)
+
+ HTRANS: hd o T(a,b) => [hd o a,hd o b] SI (DEF o T(a,b))
+
+ LTRANS: last o T(a,b) => [last o a,last o b] SI (DEF o T(a,b))
+
+ TLTRANS: t1 o T(a,b) => T(t1 o a,t1 o b)
+
+ TLIST: T(L(x),L(y)) => L([x,y])
+
+ TVID: T(<>~,<>~) => <>~
+
+ DISTL: D(x,a) o h => D(x o h,a o h)
+
+ HDISTL: hd o D(x,a) => [x,hd o a]
+
+ LDISTL: last o D(x,a) => [x,last o a]
+
+ TLDISTL: t1 o D(x,a) => D(x,t1 o a)
+
+ DLIST: D(x,L(y)) => L([x,y])
+
+ DVID: D(x,<>~) => <>~ SI (DEF o x)
+
+ HIOTA: hd o iota => 1~ SI (DEF o iota)
+
+ HIOTA': hd o (iota o n) => 1~ SI (DEF o (iota o n))
+
+ LIOTA: last o iota => id SI (DEF o iota)
+
+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))
+

```

```
+ APP: A(a,b) o h => A(a o h,b o h)
+
+ HAPP: hd o A(a,b) => hd o a SI ((DEF o A(a,b)) & (DEF o (hd o a)))
+
+ LAPP: last o A(a,b) => last o b
+
+ SI ((DEF o A(a,b)) & (DEF o (last o b)))
+
+ TLAPP: tl o A(a,b) => A(tl o a,b) SI (DEF o (tl o a))
+
+ APPASS: A(A(a,b),c) => A(a,A(b,c))
+
+ APPDISTL: D(x,A(a,b)) => A(D(x,a),D(x,b))
+
+ APPTRANS: T(A(a,b),A(c,d)) => A(T(a,c),T(b,d))
+
+ SI (DEF o (A(T(a,c),T(b,d))))
+
+ APPVID: A(a,<>) => a SI (DEF o A(a,<>))
+
+ APPVIDS: A(<>,a) => a SI (DEF o A(<>,a))
+
```

et qu'on ajoute aux axiomes sur "ALL"

```

+-----+
+ orientation manuelle: S-ALL'
+ *****
+
+ S-PRIMITIVE'
+
+ *****
+
+ ALLALL: ALL f o ALL g => ALL(f o g)
+
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h
+
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f)
+
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h))
+
+ LALL: last o ALL f => f o last SI (DEF o ALL f)
+
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (ALL f o h))
+
+ TLALL: tl o ALL f => ALL f o tl SI (DEF o ALL f)
+
+ TLALL': tl o (ALL f o h) => ALL f o (tl o h) SI (DEF o (ALL f o h))
+
+ ALLID: ALL id => id
+
+ ALLLIST: ALL f o L(x) => L(f o x)
+
+ ALLVID: ALL f o <^ => <^
+
+ FALL: T(ALL f, ALL g) => ALL[f, g]
+
+ FALL': T(ALL f o h, ALL g o h) => ALL[f, g] o h
+
+ FALLg: T(ALL f, id) => ALL[f, id]
+
+ FALLid: T(id, ALL f) => ALL[id, f]
+
+ FALL'g: T(ALL f o h, h) => ALL[f, id] o h
+
+ FALL'd: T(h, ALL f o h) => ALL[id, f] o h
+
+ TALL: T(a, a) => ALL[id, id] o a
+
+-----+

```

## REMARQUE

On a encore un système convergent avec

ALLAPP:  $\text{ALL } f \circ A(a,b) \Rightarrow A(\text{ALL } f \circ a, \text{ALL } f \circ b)$

TDISTL:  $T(\text{ALL } ?1 \circ D(x,a), a) \Rightarrow D(x,a),$

mais, pour composer le système de réécriture S-\*\*, REVE diverge si on intègre la règle ALLAPP.

#### 6. Avec les axiomes sur "•",

on construit deux systèmes de réécriture:

```

+++++
+
+ orientation manuelle: S-•
+
+ *****
+ S-ALL'
+
+ *****
+ H*': $\text{hd } \circ *f \Rightarrow ?1 \text{ SI } (\text{DEF } \circ *f)$
+
+ H*': $\text{hd } \circ (*f \circ x) \Rightarrow ?1 \circ x \text{ SI } (\text{DEF } \circ (*f \circ x))$
+
+ *VID: $*f \circ [x, <^{\sim}] \Rightarrow L(x)$
+
+ *LIST: $*f \circ [x, L(y)] \Rightarrow A(L(x), L(f \circ [x, y]))$
+
+ *CONS: $*f \circ [x, A(L(y), a)] \Rightarrow A(L(x), *f \circ [f \circ [x, y], a])$
+
+ TL*+IOTA: $\text{tl } \circ (*(\circ \circ [?1, 1^{\sim}]) \circ [0^{\sim}, \text{iota}]) \Rightarrow \text{iota}$
+
+ TL*+IOTA': $\text{tl } \circ (*(\circ \circ [?1, 1^{\sim}]) \circ [0^{\sim}, \text{iota } \circ n]) \Rightarrow \text{iota } \circ n$
+
+ L*+IOTA: $\text{last } \circ (*(\circ \circ [?1, 1^{\sim}]) \circ [0^{\sim}, \text{iota}]) \Rightarrow \text{id}$
+
+ L*+IOTA': $\text{last } \circ (*(\circ \circ [?1, 1^{\sim}]) \circ [0^{\sim}, \text{iota } \circ n]) \Rightarrow n$
+
+ *ALL: $*f \circ [x, \text{ALL } g \circ a] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, a]$
+
+ *ALL': $*f \circ [x, \text{ALL } g] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, \text{id}]$
+
+
+
+++++
```

et

```

+++++
+
+ ordonnancement manuel: S-•
+
+ *****
+ S-ALL1
+
+ *****
+ *ALL: $*f \circ [x, \text{ALL } g \circ h] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, h]$
+
+ *ALL': $*f \circ [x, \text{ALL } g] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, \text{id}]$
+
+ H*: $\text{hd } \circ (*f \circ [x, a]) \Rightarrow x \text{ SI } (\text{DEF } \circ (*f \circ [x, a]))$
+
+ F*: $\text{trans } \circ (*f \circ [x, a], *g \circ [y, a])$
+
+ $\Rightarrow *(f \circ [?1 \circ ?1, ?2], g \circ [?2 \circ ?1, ?2]) \circ [[x, y], a]$
+
+ FALL*: $\text{trans } \circ [\text{ALL } g \circ (*f \circ [x, a]), *f \circ [x, a]]$
+
+ $\Rightarrow [g \circ (f \circ [?2 \circ ?1, ?2]), f \circ [?2 \circ ?1, ?2]] \circ [[g \circ x, x], a]$
+
+ FALL*S: $\text{trans } \circ [*f \circ [x, a], \text{ALL } g \circ (*f \circ [x, a])]$
+
+ $\Rightarrow *(f \circ [?1 \circ ?1, ?2], g \circ (f \circ [?1 \circ ?1, ?2])) \circ [[x, g \circ x], a]$
+
+ F**S: $\text{trans } \circ [*f \circ [x, \text{tl } \circ (*g \circ [y, a])], *g \circ [y, a]]$
+
+ $\Rightarrow *(f \circ [?1 \circ ?1, g \circ [?2 \circ ?1, ?2]], g \circ [?2 \circ ?1, ?2]) \circ [[x, y], a]$
+
+ F**S: $\text{trans } \circ [*g \circ [y, a], *f \circ [x, \text{tl } \circ (*g \circ [y, a])]]$
+
+ $\Rightarrow [g \circ [?1 \circ ?1, ?2], f \circ [?2 \circ ?1, g \circ [?1 \circ ?1, ?2]]] \circ [[y, x], a]$
+
+
+
+++++
```

#### 7. Avec les axiomes de "WH",

on construit, comme pour les axiomes de "•", deux systèmes de réécriture:

```

+++++
+ orientation manuelle: S-WH
+ +++++
+ S-*
+ +++++
+ WHALL: WH(p,f) o [x,ALL g o h]
+
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,h]
+
+ WHALL': WH(p,f) o [x,ALL g]
+
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,id]
+
+ WHVID: WH(p,f) o [x,<>] => L(x)
+
+ WHLIST1: WH(p,f) o [x,L(y)] => L(x) SI (not o (p o [x,y]))
+
+ WHLIST2: WH(p,f) o [x,L(y)] => A(L(x),L(f o [x,y])) SI (p o [x,y])
+
+ WHCONS: WH(p,f) o [x,A(L(y),h)] =>
+
+ A(L(x),WH(p,f) o [f o [x,y],h]) SI (p o [x,y])
+
+ HWH: hd o WH(p,f) => ?1 SI (DEF o WH(p,f))
+
+ HWH': hd o (WH(p,f) o z) => ?1 o z SI (DEF o (WH(p,f) o z))
+
+++++

```

et

```

+++++
+ ordonnancement manuel: S-WHWH
+ +++++
+ S-*
+ +++++
+ WHALL: WH(p,f) o [x,ALL g o a]
+ => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,a]
+
+ WHALL': WH(p,f) o [x,ALL g] => WH(p o [?1,g o ?2],f o [?1,g o ?2]) o [x,id]
+
+ HWH: hd o (WH(p,f) o [x,a]) => x SI (DEF o (WH(p,f) o [x,a]))
+
+ ALLWH: trans o [ALL g o (WH(p,f) o [x,a]),WH(p,f o [x,a])]
+
+ => WH(p o [?2o?1,?2],[g o (f o [?2o?1,?2]),f o [?2o?1,?2]]) o [[g o x,x],a]
+
+ ALLWHS: trans o [WH(p,f) o [x,a],ALL g o (WH(p,f) o [x,a])]
+
+ => WH(p o [?1o?1,?2],[f o [?1o?1,?2],g o (f o [?1o?1,?2])]) o [[x,g o x],a]
+
+ *WH: trans o [*f o [x,t1 o (WH(p,g) o [y,a]),WH(p,g) o [y,a]] =>
+
+ WH(p o [?2o?1,?2],[f o [?1o?1,g o [?2o?1,?2]],g o [?2o?1,?2]) o [[x,y],a]
+
+ *WHS: trans o [WH(p,g) o [y,a],*f o [x,t1 o (WH(p,g) o [y,a])]] =>
+
+ WH(p o [?1o?1,?2],[g o [?1o?1,?2],f o [?2o?1,g o [?1o?1,?2]]) o [y,x],a]
+
+++++

```

8. Avec les axiomes de " $\_ \rightarrow \_ ; \_$ ",

on construit les systèmes de réécriture convergents :

```

+++++
+ S-CONDO +
+ *****+
+ TOUT SYSTEME +
+ *****+
+ COND1: (p->f;g) o h => (p o h)->(f o h);(g o h) +
+ COND51: p->[h,f];[h,g] => [h,p->f;g] +
+ COND52: p->[f,h];[g,h] => [p->f;g,h] +
+ S: p->f;f => f SI (DEF o (p->f;f)) +
+ T: (p->x;y)->f;g => p->(x->f;g);(y->f;g) +
+ TRUE: T-->x;y => x +
+ FALSE: F-->x;y => y +
+++++

```

```

+++++
+ S-COND1 +
+ *****+
+ TOUT SYSTEME +
+ *****+
+ COND1: (p->f;g) o h => (p o h)->(f o h);(g o h) +
+ COND3: p->(p->f;g);h => p->f;h +
+ COND4: p->h;(p->f;g) => p->h;g +
+ S: p->f;f => f SI (DEF o (p->f;f)) +
+ TRUE: T-->f;g => f +
+ FALSE: F-->f;g => g +
+ PPX: p->p;x => p->T-;x +
+ PXP: p->x;p => p->x;F- +
+ TF: p->T-;F- => p +
+++++

```

```

+++++
+ S-COND2 +
+ *****+
+ TOUT SYSTEME +
+ *****+
+ COND1: (p->f;g) o h => (p o h)->f o h);(g o h) +
+ COND51: p->[h,f];[h,g] => [h,p->f;g] +
+ COND52: p->[f,h];[g,h] => [p->f;g,h] +
+ S: p->h;h => h SI (DEF o (p->h;h)) +
+ TRUE: T-->x;y => x +
+ FALSE: F-->x;y => y +
+ Tf: p->T-;F- => p +
+ PPX: p->p;x => p->T-;x +
+ PPY: p->x;p => p->x;F- +
+++++

```

### ANNEXE III : LES META-SYSTEMES DE REECRITURE CONVERGENTS

En utilisant le formalisme des méta-règles, nous avons isolé des systèmes de réécriture convergents qui intègrent les familles infinies d'axiomes de la FP-algèbre.

#### REMARQUES

Nous notons les noms des méta-règles en caractères gras.

A coté de la méta-règle, on fait référence au domaine de ces méta-variables et au système de réécriture  $R_0$  qui lui est associé.

Les définitions des domaines et les règles des systèmes de réécriture  $R_0$  sont données à la suite de chaque système de réécriture.

Pour définir les domaines des méta-variables:

- $T(S,V)$  est l'ensemble des FP-termes.
- Les ensembles de termes qui servent à définir le domaine des méta-variables sont définis par des équations récursives. Si  $E$  et  $F$  sont deux ensembles de termes et si "OP" est un symbole d'arité 2,

$$E \text{ OP } F = \{x \text{ OP } y \text{ t.q. } x \in E \text{ et } y \in F.\}$$

#### 1. Avec les axiomes E1 et E2

```

+++++
+ S-E1E2 +
+ ***** +
+ E1: (f o g) o h => f o (g o h) +
+ E2: [f,G] o h => [f o h,G o h] +
+ +
+++++

```

$R_0 = \{ASS, DIST\}$

$G \in \text{DOM}$

$$\text{DOM} = T(S,V) \cup (\text{DOM}.\text{DOM})$$

$$\text{ASS: } (x.y).z \Rightarrow x.(y.z)$$

$$\text{DIST: } (x.y) \circ h \Rightarrow (x \circ h).(y \circ h)$$

#### 2. Avec les axiomes de base

```

+++++
+ S-BASE +
+ ***** +
+ S-E1E2 et +
+ ***** +
+ E3: x~ o y => x~ SI (DEF o y) +
+ P1: id o f => f +
+ P2: f o id => f +
+ P3: ?I^N o [f;G] => ?(I-1)(N-1){G} +
+ ?1^N o [f;G] => f SI (DEF o [f;G]) +
+ +
+++++

```

$G \in \text{LG}$

$I = 1, 2, \dots \text{ t.q. } I \leq N$   
 $N = 2, 3, \dots$

$R_0 = \{VG\}$

$$\text{LG} = \{\text{NIL}\} \cup (T(S,V) : \text{LG})$$

$$\text{VG: } x : \text{NIL} \Rightarrow x$$

#### 3. Avec les axiomes sur les primitives sur les séquences,

```

+-----+
+ S-PRIMITIVE
+-----+
+
+ S-BASE et
+
+ *****
+
+ LTL: last o t1 => last SI (DEF o (last o t1))
+
+ LTL': last o (t1 o h) => last o h SI (DEF o (last o (t1 o h)))
+
+ LIST: L(x) o y => L(x o y)
+
+ HLIST: hd o L(x) => x
+
+ LCVID: last o L(x) => x
+
+ TLLIST: t1 o L(x) => <>~ SI (DEF o x)
+
+ HTRANS: hd o (trans o [a,B]) => [hd o a,hd o B] B ∈ DOM
+
+ SI (DEF o (trans o [a,B])) RO={ASS,HD}
+
+ LTRANS: last o (trans o [a,B]) => [last o a,last o B]
+
+ SI (DEF o (trans o [a,B])) RO={ASS,LAST}
+
+ TLTRANS: t1 o (trans o [a,B]) => trans o [t1 o a,t1 o B]
+
+ RO = {ASS,TL}
+
+ TLIST: trans o [L(x),L(y)] => L([x,y]) Y ∈ DOM, RO={ASS,L}
+
+ HDISTL: hd o (distl o [x,a]) => [x,hd o a]
+
+ LDISTL: last o (distl o [x,a]) => [x,last o a]
+
+ TLDISTL: t1 o (distl o [x,a]) => distl o [x,t1 o a]
+
+ DLIST: distl o [x,L(y)] => L([x,y])
+
+ TVID: trans o [V] => <>~ V ∈ LV et RO={ASS}
+
+ DVID: distl o [x,<>~] => <>~ SI (DEF o x)
+
+ HIOTA: hd o iota => 1~ SI (DEF o iota)
+
+ HIOTA': hd o (iota o n) => 1~ SI (DEF o (iota o n))
+
+ LIOTA: last o iota => id SI (DEF o iota)
+
+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))

```

```
+ HAPP: hd o (append o [a,b]) => hd o a
+
+ SI ((DEF o (hd o a)) & (DEF o (append o [a,b])))
+
+
+ LAPP: last o (append o [a,b]) => last o b
+
+ SI ((DEF o (last o b)) & (DEF o (append o [a,b])))
+
+
+ TLAPP: tl o (append o [a,b]) => append o [tl o a,b] SI (DEF o (tl o a))
+
+ DEFCONS: cons o [x,a] => append o [L(x),a]
+
+ APPASS: append o [append o [a,b],c] => append o [a,append o [b,c]]
+
+ APPDISTL: distl o [x,append o [a,b]]
+ => append o [distl o [x,a],distl o [x,b]]
+
+ APPTRANS: trans o [append o [A,B],append o [C,D]] A et B ∈ T(S,V)
+
+ => append o [trans o [A,C],trans o [B,D]] C et D ∈ DOM
+
+ SI ((DEF o [trans o [A,C],trans o [B,D]]) RO={APPASS,ASS,APP})
+
+
+ APPVID: append o [a,<-] => a SI (DEF o (append o [a,<-]))
+
+ APPVIDS: append o [<- ,a] => a SI (DEF o (append o [<- ,a]))
```

a) Pour HTRANS, LTRANS et TLTRANS

$$\text{DOM} = T(S, V) \cup (\text{DOM}.\text{DOM})$$

ASS:  $(x.y).z \Rightarrow x.(y.z)$

$$\text{HD: } \text{hd } o \ (x.y) \Rightarrow (\text{hd } o \ x).(\text{hd } o \ y)$$

LAST: last o (x.y) => (last o x).(last o y)

$$TL: tl \circ (x.y) \Rightarrow (tl \circ x).(tl \circ y)$$
$$L: L(x.y) \Rightarrow L(x).L(y)$$

b) Pour TVID

$$LV = \langle \rangle^{\sim} \cup (LV.LV)$$

ASS:  $(x.y).z \Rightarrow x.(y.z)$

c) Pour APPTANS

DOM=T(S,V)U (DOM.DOM)

ASS:  $(x.y).z \Rightarrow x.(y.z)$

APP:  $\text{append } o [x.y,u.v] \Rightarrow (\text{append } o [x,u]).(\text{append } o [y,v])$

APPASS:  $\text{append } o [\text{append } o [a,b],c] \Rightarrow \text{append } o [a,\text{append } o [b,c]]$

4. Avec les axiomes de "ALL"

```

+++++
+ S-ALL +
+ *****+
+ S-PRIMITIVE et +
+ *****+
+
+ ALLALL: ALL f o ALL g => ALL(f o g) +
+
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h +
+
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f) +
+
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h)) +
+
+ LALL: last o ALL f => f o last SI (DEF o ALL f) +
+
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (last o h)) +
+
+ TLALL: tl o ALL f => ALL f o tl SI (DEF o ALL f) +
+
+ TLALL': tl o (ALL f o h) => ALL f o (tl o h) +
+
+ SI (DEF o (ALL f o h)) +
+
+ ALLID: ALL id => id +
+
+ ALLLIST: ALL f o L(x) => L(f o x) +
+
+ ALLAPP: ALL f o (append o [a,b]) => append o [ALL f o a,ALL f o b] +
+
+ ALLVID: ALL f o <> => <> +
+
+ FALL: trans o [ALL F,ALL G] => ALL[F,G] F et G ∈ DOM +
+
+ RO = {ASS,ALL,ALLID} +
+
+ FALL': trans o [ALL F o h,ALL G o h] => ALL[F,G] o h +
+
+ RO={ASS,ALL,DIST,ALLID} +
+
+ TDISTL: trans o [ALL?l o (distl o [x,a]),a] => distl o [x,a] +
+
+++++

```

DOM= T(S,V) U (DOM.DOM)



```

+++++
+ S-PRIMITIVE'
+ *****
+ S-BASE'
+ *****
+ LTL: last o t1 => last SI (DEF o (last o t1))
+
+ LTL': last o (t1 o h) => last o h SI (DEF o (last o (t1 o h)))
+
+ LIST: L(x) o y => L(x o y)
+
+ HLIST: hd o L(x) => x
+
+ LLIST: last o L(x) => x
+
+ TLLIST: t1 o L(x) => <> SI (DEF o x)
+
+ TRANS: T(a,B) o h => T(a o h, B o h) B ∈ DOM
+
+ RO={ASS,DIST}
+
+ HTRANS: hd o T(a,B) => [hd o a, hd o B] SI (DEF o T(a,B))
+
+ RO={ASS,HD}
+
+ LTRANS: last o T(a,B) => [last o a, last o B] SI (DEF o T(a,B))
+
+ RO={ASS,LAST}
+
+ TLTRANS: t1 o T(a,B) => T(t1 o a, t1 o B) RO={ASS,TL}
+
+ TLIST: T(L(x), L(y)) => L([x,y]) Y ∈ DOM, RO={ASS,L}
+
+ TVID: T(<>, V) => <> V ∈ LV et RO={ASS}
+
+ DISTL: D(x,a) o h => D(x o h, a o h)
+
+ HDISTL: hd o D(x,a) => [x, hd o a]
+
+ LDISTL: last o D(x,a) => [x, last o a]
+
+ TLDISTL: t1 o D(x,a) => D(x, t1 o a)
+
+ DLIST: D(x, L(y)) => L([x,y])
+
+ DVID: D(x, <>) => <> SI (DEF o x)
+
+ HIOTA: hd o iota => 1 SI (DEF o iota)
+
+ HIOTA': hd o (iota o n) => 1 SI (DEF o (iota o n))

```

```

+
+ LIOTA: last o iota => id SI (DEF o iota)
+
+ LIOTA': last o (iota o n) => n SI (DEF o (iota o n))
+
+ APP: A(a,b) o h => A(a o h, b o h)
+
+ HAPP: hd o A(a,b) => hd o a SI ((DEF o A(a,b)) & (DEF o (hd o a)))
+
+ LAPP: last o A(a,b) => last o b
+
+ SI ((DEF o A(a,b)) & (DEF o (last o b)))
+
+ TLAPP: t1 o A(a,b) => A(t1 o a, b) SI (DEF o (t1 o a))
+
+ APPASS: A(A(a,b), c) => A(a, A(b,c))
+
+ APPDISTL: D(x, A(a,b)) => A(D(x,a), D(x,b))
+
+ APPTRANS: T(A(X,Y), A(U,V)) => A(T(X,U), T(Y,V)) X et Y ∈ T(S,V)
+
+ SI (DEF o (A(T(X,U), T(Y,V)))) U et V ∈ DOM
+
+ RO={ASS,APP,APPASS}
+
+ APPVID: A(a, <>) => a SI (DEF o A(a, <>))
+
+ APPVIDS: A(<>, a) => a SI (DEF o A(<>, a))
+
+
+++++

```

a) Pour TRANS, HTRANS, LTRANS, TLTRANS et TVID,

les définitions de DOM et LV, les axiomes ASS, HD, LAST, TL et L sont ceux que l'on a donnés pour S-PRIMITIVE. DIST est l'axiome donné avec S-BASE.

b) Pour APPASS

L'axiome ASS est le même que dans S-PRIMITIVE,

APP:  $A(x.y, u.v) \Rightarrow A(x, u).A(y, v)$

et qu'on ajoute aux axiomes sur "ALL"

```

+++++
+ S-ALL'
+ *****
+
+ S-PRIMITIVE'
+
+ *****
+
+ ALLALL: ALL f o ALL g => ALL(f o g)
+
+ ALLALL': ALL f o (ALL g o h) => ALL(f o g) o h
+
+ HALL: hd o ALL f => f o hd SI (DEF o ALL f)
+
+ HALL': hd o (ALL f o h) => f o (hd o h) SI (DEF o (ALL f o h))
+
+ LALL: last o ALL f => f o last SI (DEF o ALL f)
+
+ LALL': last o (ALL f o h) => f o (last o h) SI (DEF o (ALL f o h))
+
+ TLALL: tl o ALL f => ALL f o tl SI (DEF o ALL f)
+
+ TLALL': tl o (ALL f o h) => ALL f o (tl o h) SI (DEF o (ALL f o h))
+
+ ALLID: ALL id => id
+
+ ALLLIST: ALL f o L(x) => L(f o x)
+
+ ALLVID: ALL f o <> => <>
+
+ FALL: T(ALL F, ALL G) => ALL[F,G] F et G ∈ DOM
+
+ RO={ASS,ALL,ALLID}
+
+ FALL': T(ALL F o h, ALL G o h) => ALL[F,G] o h
+
+ RO={ASS,ALL,DIST,ALLID}
+++++

```

Toutes les définitions se trouvent avec S-ALL.

6. Avec les axiomes sur "\*",

```

+++++
+ S-***
+ *****
+
+ S-ALL1
+
+ *****
+
+ *ALL: *f o [x,ALL g o h] => *(f o [?1,g o ?2]) o [x,h]
+
+ *ALL': *f o [x,ALL g] => *(f o [?1,g o ?2]) o [x,id]
+
+ H*: hd o (*f o [x,a]) => x SI (DEF o (*f o [x,a]))
+
+ F*: trans o [*F o [X,a]:*f o [y,a]] X et F ∈ LD
+
+ => *[(F o [K(L(F)-1,L(F)),?2L(F)])RO={VD,**,NIL,DL,DLNIL,L1,L2,K1,K2}+
+
+ : (g o [?nL(F)o?1L(F)])] o [[X:y],a]
+
+ FALL*: trans o [ALL g o (*f o [x,a]),*f o [x,a]]
+
+ => *[g o (f o [?2o?1,?2]),f o [?2o?1,?2]] o [[g o x,x],a]
+
+ FALL*S: trans o [*f o [x,a],ALL g o (*f o [x,a])
+
+ => *[f o [?1o?1,?2],g o (f o [?1o?1,?2])] o [[x,g o x],a]
+
+ F**S: trans o [*f o [x,t1 o (*g o [y,a]),*g o [y,a]]
+
+ => *[f o [?1o?1,g o [?2o?1,?2]],g o [?2o?1,?2]] o [[x,y],a]
+
+ F**S: trans o [*g o [y,a],*f o [x,t1 o (*g o [y,a])
+
+ => *[g o [?1o?1,?2],f o [?2o?1,g o [?1o?1,?2]]] o [[y,x],a]
+++++

```

LD= T(S,V) U (LD:LD)

VD: NIL:x => x

\*\*S: \*(x:y) => \*x:\*y

\*NIL: \*NIL => NIL

DL: x:y o [u:v,h] => (x o [u,h]):(y o [u,h])

DLNIL: NIL o [NIL,h] => NIL

$L1: L(NIL) \Rightarrow 1$   
 $L2: L(x:y) \Rightarrow L(x) + 1$   
 $K1: K(1,n) \Rightarrow ?1^n o ?1^n$   
 $K1: K(p+1,n) \Rightarrow K(p,n):(?p^n o ?1^n)$

7. Avec les axiomes de " $\_ \rightarrow \_ ; \_$ ",

Dans les systèmes S-CONDO et S-COND2 (voir l'annexe 2), la règle COND5 s'étend en une méta-règle COND5

COND5:  $p \rightarrow [H1.f.H2]; [H1.g.H2] \Rightarrow [H1.p \rightarrow f; g.H2]$

$H1$  et  $H2 \in \text{DOMV}$   $RO = \{ASS, NILg, NILd\}$

$\text{DOMV} = NIL \cup T(S,V) \cup (\text{DOMV}.\text{DOMV})$

ASS:  $(x.y).z \Rightarrow x.(y.z)$

NILg:  $NIL.x \Rightarrow x$  et NILd:  $x.NIL \Rightarrow x$

ANNEXE 4: DES EXEMPLES

Nous appliquons l'ordre REVE "NORMAL FORM" aux expressions fonctionnelles codant nos exemples du chapitre I avec le système convergent S-\*\*. Nous numérotons les exemples et les expressions comme dans le chapitre I.

1. Exemple 1: calcul de  $2^n$

(1)  $\text{last } o \ ( *mult \ o \ [1^-, ALL \ 2^- \ o \ (iota \ o \ n)])$

Nous comptons deux séquences intermédiaires résultats de "ALL 2<sup>-</sup>" et de "\*mult":  $N(1) = I(1) = 2$ .

$(1) \downarrow_{S-**} = (2)$

(2)  $\text{last } o \ ( *mult \ o \ [?1, 2^-]) \ o \ [1^-, iota \ o \ n]$

L'expression (2) ne spécifie qu'une séquence intermédiaire. La séquence de 2 qui est introduite par "ALL 2<sup>-</sup>" dans l'expression (1) a disparu:  $N(2) = I(2) = 1$ .

2. Exemple 2: calcul de  $x^n$

x est "bien placé"

(1)  $\text{last } o \ ( *mult \ o \ [1^-, ALL \ ?1 \ o \ (dist1 \ o \ [x, iota \ o \ n])])$

On compte  $N(1) = 3$  et  $I(1) = 2$ .

$(1) \downarrow_{S-**} = (3)$

(3) last o (\*g' o [1<sup>-</sup>, distl o [x, iota o n]])

where def g' = mult o [?1, ?1o?2]

On compte N(3) = 2 et I(3) = 1.

x est "mal placé"

(2) last o (\*g o [1<sup>-</sup>, distl o [id, iota o n]])

where def g = mult o [?1, x o (?1o?2)]

On compte N(2) = 2 et I(2) = 1.

(2)<sub>S-\*</sub> = (2)

L'axiome ALLDISTL ajouté à S-\*

ALL[f o ?1, g o ?2] o (distl o [x, a]) = distl o [f o x, ALL g o a]

provoque une divergence de REVE. En rejetant des paires critiques et en laissant poursuivre la complétion, on réussit à obtenir la nouvelle règle

\*(f o [?1, [g o (?1 o ?2), ?2 o ?2]]) o [x, distl o [y, a]]

=> \*f o [x, distl o [g o y, iota o n]]

Si on interrompt la complétion, l'ordre "normal form" réduit le terme

(2) dans l'expression (3) précédente.

3. Exemple 3: calcul de  $2^n/n!$

Les deux expressions "les plus mauvaises"

(1) last o (ALL / o (trans o [u, v]))

where def u = \*mult o [1<sup>-</sup>, ALL 2<sup>-</sup> o (iota o n)]

def v = \*mult o [1<sup>-</sup>, iota o n]

(on compte N(1) = 5 et I(1) = 4) et

(2) / o (last o (trans o [u', v]))

where def u' = \*(mult o [?1, 2<sup>-</sup>]) o [1<sup>-</sup>, iota o n]

(on compte N(2) = 3 et I(2) = 2) sont toutes deux réduites en

(5) / o (last o (\*g o [[1<sup>-</sup>, 1<sup>-</sup>], iota o n])

where def g = [ mult o [2<sup>-</sup>, ?1o?1] , mult o [?2o?1, ?2] ]

On compte N(5) = I(5) = 1.

(1)<sub>S-\*</sub> = (2)<sub>S-\*</sub> = (5)

Mais l'expression

(3) / o [last o u, last o v]

(on compte N(3) = I(3) = 3) est réduite en

(3)<sub>S-\*</sub> / o [last o u', last o v]

pour laquelle on compte encore deux séquences intermédiaires.

La règle LTL

last o (trans o [a, b]) => [last o a, last o b]

SI (DEF o (last o (trans o [a, b]))

provoque, dans le système S-\*, une divergence de REVE. Mais avant de constater la divergence, on obtient la nouvelle règle LF\*\*

[last o (\*f o [x, a]), last o (\*g o [y, a])

=> last o (\*[f o [?1 o ?1, ?2], g o [?2 o ?1, ?2]] o [[x, y], a])

Si on interrompt alors la complétion, l'expression (3) se réduit aussi dans l'expression (5).

L'expression

(4) last o (\*g o [1<sup>-</sup>,iota o n])

where def g = mult o [?1,/ o [2<sup>-</sup>,?2]]

pour laquelle, on compte, comme pour l'expression (5)  $N(4) = I(4) = 1$ , est irréductible. Elle n'est pas considérée comme équivalente à l'expression (5). L'expression (4) ne peut être la forme normale des expressions précédentes que si l'on met en jeu les axiomes de "mult" et de "/" et la règle R2 du chapitre II.

#### 4. Exemple 4:

L'expression du calcul de  $\Sigma a_i b_i + \Sigma c_i d_i$

(e) + o (last o (trans o (sps o [ALL a o (iota o n), ALL b o (iota o n)]  
sps o [ALL c o (iota o n), ALL d o (iota o n)]

where def sps = \*\* o [0<sup>-</sup>, ALL mult o trans]

pour laquelle on compte  $N(e)=11$  et  $I(e) = 9$ , a pour forme normale

(e)<sub>S-\*</sub> + o (last o (\*[+ o [?1 o ?1, mult o [a o ?2, b o ?2]],  
+ o [?2 o ?1, mult o [c o ?2, d o ?2]])  
o [[0<sup>-</sup>, 0<sup>-</sup>],iota o n]))

pour la quelle on ne compte qu'une séquence intermédiaire.

L' expression

(1) + o [ps o [ALL a o (iota o n), ALL b o (iota o n)],  
ps o [ALL c o (iota o n), ALL d o (iota o n)]]

where def ps = last o sps

pour laquelle on compte  $N(1) = 10$  et  $I(1) = 8$ , a pour forme normale

(1)<sub>S-\*</sub> + o [ps<sub>1</sub>, ps<sub>2</sub>] where  
def ps<sub>1</sub> = last o (\*[+ o [?1, mult o [a o ?2, b o ?2]]) o [0<sup>-</sup>,iota o n])  
def ps<sub>2</sub> = last o (\*[+ o [?1, mult o [c o ?2, d o ?2]]) o [0<sup>-</sup>,iota o n])

pour laquelle on compte 2 séquences intermédiaires.

Pour que la forme normale de (1) soit égale à celle de (e), il faut, comme pour l'exemple précédent, ajouter à S-\*\* l'axiome LTL et arrêter la complétion quand on obtient la nouvelle règle LF\*\*.

Nous pouvons réduire "manuellement" ces expressions en

(2) last o (\*f o [0<sup>-</sup>,iota o n])

where def f = + o [?1

,+ o [ mult o [a o ?2, b o ?2]

, mult o [c o ?2, d o ?2] ] ]

qui cumule les calculs parallèles des aibi et des cidi pour i de 1 à n en utilisant les axiomes de "+" et de "mult" ainsi que la règle R2 du chapitre II.

#### 5. Exemple 5:

L'expression du calcul de  $\Sigma 2^i$

(i) last o (\*g o [0<sup>-</sup>,iota o n])

where def g = + o [?1, f o ?2]

and def f = last o (\*mult o [1<sup>-</sup>, ALL 2<sup>-</sup> o iota] )

compte  $I(i) = N(i) = 1 + 2*n$ .

La forme normale est

(i)<sub>S-\*</sub> last o (\*g' o [0<sup>-</sup>,iota o n])

where def g = + o [?1, u]

and def u = last o (\*[mult o [?1, 2<sup>-</sup>]] o [1<sup>-</sup>,iota o ?2]))

et l'on compte n + 1 séquences intermédiaires.

Si on ajoute à S-\*\* l'axiome ALL\*IOTA

$ALL(last \circ (*f \circ [g \circ x^-,iota])) \circ iota \Rightarrow t1 \circ (*f \circ [g \circ x^-,iota])$

REVE diverge. Mais si l'on continue la complétion, on obtient la règle

$*(k \circ [?1, last \circ (*f \circ [x^-,iota \circ ?2])]) \circ [y,iota \circ n]$   
 $\Rightarrow *k \circ [y,t1 \circ (*f \circ [x^-,iota \circ n])].$

Si l'on arrête la complétion, (i) est réduite en

$last \circ (*+ \circ [0^-,t1 \circ (*mult \circ [2^-,iota \circ n])])$

et l'on compte 2 séquences intermédiaires.

#### 6. Exemple 6: des calculs de moyennes

##### L'expression

(1)  $ALL (/ \circ [s_1, s_2]) \circ (iota \circ n)$   
 where def  $s_2 = last \circ (*+ \circ [0^-,u])$   
 and def  $u = ALL p \circ (iota \circ 10^-)$   
 where def  $s_1 =$   
 $last \circ (*+ \circ [0^-,$   
 $ALL mult \circ (trans$   
 $\circ [ALL p \circ (iota \circ 10^-),$   
 $ALL x \circ (dist1 \circ [id, iota \circ 10^-])])]$

compte:  $N(1) = 8*n + 1$  et  $I(1) = 6*n + 1$ .

La forme normale est

(1) $\downarrow_{S-**}$   $ALL (/ \circ [u_1, u_2]) \circ (iota \circ n)$   
 where def  $u_2 = last \circ (*+ \circ [?1, p \circ ?2]) \circ [0^-,iota \circ 10^-]$   
 def  $u_1 = last \circ (*+ \circ [?1, mult \circ ?2])$   
 $\circ [0^-,trans \circ [ALL p \circ (iota \circ 10^-),$   
 $ALL x \circ (dist1 \circ [id, iota \circ 10^-])]]]$

et compte  $6*n + 1$  pour N et  $4*n + 1$  pour I.

##### L'expression

(2)  $ALL (/ \circ [?2, ?1]) \circ (dist1 \circ [s_2, ALL s_1 \circ (iota \circ n)])$   
 compte  $N(2) = 6*n + 5$  et  $I(2) = 4*n + 4$  et a pour forme normale  
 (2) $\downarrow_{S-**}$   $ALL (/ \circ [?2, ?1]) \circ (dist1 \circ [u_2, ALL u_1 \circ (iota \circ n)])$   
 qui compte  $5*n + 4$  pour N et  $3*n + 3$  pour I.

##### L'expression

(3)  $ALL (/ \circ [s_1 \circ ?2, ?1]) \circ (dist1 \circ [s_2, iota \circ n])$

compte  $N(3) = 6*n + 4$  et  $I(3) = 4*n + 3$  et est réduite en

(3) $\downarrow_{S-**}$   $ALL (/ \circ [u_1 \circ ?2, ?1]) \circ (dist1 \circ [u_2, iota \circ n])$

qui compte  $5*n + 3$  pour N et  $3*n + 2$  pour I.

Si l'on ajoute l'axiome ALLDISTL

$ALL[f \circ ?1, g \circ ?2] \circ (dist1 \circ [x, a]) \Rightarrow dist1 \circ [f \circ x, ALL g \circ a]$

REVE diverge, mais l'on peut arrêter la complétion quand on obtient la nouvelle règle

$ALL k \circ (dist1 \circ [x, ALL g \circ a]) \Rightarrow ALL(k \circ [?1, g \circ ?2]) \circ (dist1 \circ [x, a]).$

(2) a alors pour forme normale (3) $\downarrow_{S-\ast}$ .

### 7. Exemple 7: moyenne simple de 10 entiers

L'expression

(1)  $?1 \circ (\text{last} \circ \ast f \circ [[0^-, 0^-], x])$

where def  $f = [ / \circ [g, 10^-], g ]$

and def  $g = + \circ [?2 \circ ?1, ?2]$

pour laquelle on compte  $N(1) = I(1) = 1$  est irréductible pour  $S-\ast$ .

L'expression

(2)  $\text{last} \circ (\text{ALL} / \circ u)$

where def  $u = \text{trans} \circ [t1 \circ (\ast + \circ [0^-, x]), \text{iota} \circ 10^-]$

est réduite en

(2) $\downarrow_{S-\ast} / \circ (\text{last} \circ u)$

et si on ajoute L7L ou si l'on prend le système  $S-\ast$ , elle est réduite en

(3)  $/ \circ [ \text{last} \circ (\ast + \circ [0^-, x]), 10^- ]$

On compte:

$N(2) = 4$  et  $I(2) = 2$

$N(3) = 1$  et  $I(3) = 1$

### 8. Exemples 8 et 9: La réduction des conditionnelles

Les expressions données au chapitre I sont irréductibles. Une réduction ne peut être faite que si l'on ajoute les axiomes de " $\_ \rightarrow \_ ; \_$ " et que l'on réécrit "modulo l'axiome C".

## ANNEXE 5: PREUVE DE TERMINAISON

La définition de l'ordre des termes est celle qui est proposée au chapitre I de la deuxième partie. Nous utilisons aussi, au cours de cette preuve, le principe d'utilisation de motifs et le principe de distinction des opérateurs que nous avons définis dans ce même chapitre.

### 1. Systèmes $S\text{-BASE}$ et $S\text{-BASE}'$

#### 1.1. Règles E1 et E2

E1:  $(f \circ g) \circ h \Rightarrow f \circ (g \circ h)$

E2:  $[f, g] \circ h \Rightarrow [f \circ h, g \circ h]$

● On définit l'interprétation I:

$I(x \circ y) = I(x) I(y)$  et  $I([x, y]) = I(x) + I(y)$

● On définit l'ordre sur les symboles:

$[ \_ ] > \circ$

On a

-  $I((f \circ g) \circ h) = I(f \circ (g \circ h)) = I(f)I(g)I(h)$  et on applique le cas 2.b de la définition de l'ordre des termes.

-  $I([f, g] \circ h) = I([f \circ h, g \circ h]) = I(f)I(h) + I(g)I(h)$  et on applique le 2.a de la définition de l'ordre des termes.

### 1.2. Règles restantes des systèmes S-BASE

E3:  $x \circ y \Rightarrow y$  SI (DEF o y)

P1:  $id \circ f \Rightarrow f$

P2:  $f \circ id \Rightarrow f$

P3:  $?1 \circ [f, g] \Rightarrow f$  SI (DEF o [f, g])

et  $?2 \circ [f, g] \Rightarrow g$  SI (DEF o [f, g])

ID:  $[?1, ?2] \Rightarrow id$  SI (DEF o [?1, ?2])

ID':  $[?1 \circ f, ?2 \circ f] \Rightarrow f$  SI (DEF o [?1 o f, ?2 o f])

● On définit I

$I(x^{-}) = 2$  pour tout objet x et

$I(c) = 2$  pour toute constante c.

Pour chaque règle l  $\Rightarrow$  r ci-dessus on trouve  $I(l) > I(r)$ .

Ce qui achève la preuve de terminaison des systèmes de réécriture S-BASE

### 2. Systèmes S-ALL

#### 2.1. Règles ALLALL et ALLALL'

ALLALL:  $ALL f \circ ALL g \Rightarrow ALL(f \circ g)$

ALLALL':  $ALL f \circ (ALL g \circ h) \Rightarrow ALL(f \circ g) \circ h$ .

● On définit I

$I(ALL x) = n I(x) + 1$

n est une constante entière strictement positive quelconque.

On a

$$I(ALL f \circ ALL g) = (n I(f) + 1) (n I(g) + 1) \\ = n^2 I(f) I(g) + n (I(f) + I(g)) + 1$$

$$> I(ALL(f \circ g)) = n I(f \circ g) + 1 = n I(f) I(g) + 1$$

en conséquence

$$I(ALL f \circ (ALL g \circ h)) = I(ALL f \circ ALL g) I(h)$$

$$> I(ALL(f \circ g) \circ h) = I(ALL(f \circ g)) I(h).$$

#### 2.2. Règle d'associativité de "append"

APPASS:  $append \circ [append \circ [a, b], c] \Rightarrow append \circ [a, append \circ [b, c]]$ .

Appelons l et r les membres droits et gauches, quelle que soit  $I(append)$ , on aura  $I(l) = I(r)$ .

Les OP-termes  $T(OP, V)$  tels que:

$$OP = (S - \{trans, distl, append, cons\}) \cup \{T, D, A, L\}$$

ont déjà été utilisés et définis au chapitre 3 de la première partie.

Nous allons utiliser le morphisme  $\Phi$  des FP-termes dans les OP-termes tel que

$$\Phi(append \circ [x, y]) = A(x, y)$$

$$\Phi(distl \circ [x, a]) = D(x, a)$$

$$\Phi(trans \circ [x, y]) = T(x, y).$$

$\Phi$  transforme APPASS en

$$A(A(a, b), c) = A(a, A(b, c)).$$

● Avec l'interprétation de "A"

$$I(A(x,y)) = I(x) + I(y)$$

on a

$$I(A(A(a,b),c)) = I(A(a,A(b,c)))$$

mais on peut conclure que  $A(A(a,b),c) > A(a,A(b,c))$  à cause du cas 2.b de la définition de l'ordre des termes.

Le "motif"  $m = \text{append } o [x,y]$  superposé sur E1 donne la règle APP

$$\text{APP: } A(a,b) \circ h \Rightarrow A(a \circ h, b \circ h).$$

● On complète l'ordre des symboles avec  $\circ > A$  et on applique le cas 2.a de la définition de l'ordre des termes.

### 2.3. Règles TLALL et TLALL'

$$\text{TLALL: } t1 \circ \text{ALL } f \Rightarrow \text{ALL } f \circ t1 \text{ SI (DEF } \circ \text{ALL } f)$$

$$\text{TLALL': } t1 \circ (\text{ALL } f \circ h) \Rightarrow \text{ALL } f \circ (t1 \circ h) \text{ SI (DEF } \circ (\text{ALL } f \circ h)).$$

Appelons  $l$  et  $r$  les deux membres des règles. Quelle que soit l'interprétation que l'on donne à "t1", on aura  $I(l) = I(r)$  et  $l$  et  $r$  seront incomparables.

Nous voulons pouvoir mesurer, par l'interprétation, le fait que ALL  $f$  s'applique à gauche à une séquence plus longue qu'à droite.

Nous considérons les N-termes qui distinguent les symboles

$\text{ALL}_n$  pour  $n=1,2,\dots$

$\Delta$  substitue, dans un N-terme " $\text{ALL}_n f$ ", " $\text{ALL } f$ " où  $n$  est la longueur de

la séquence à laquelle "ALL  $f$ " s'applique. Il peut faire correspondre une infinité de N-termes à un OP-terme.

$\Delta^{-1}$  donne des familles infinies de règles dans les N-termes. Par exemple, ALLALL et ALLALL' sont des familles infinies de règles:

$$\text{ALL}_n f \circ \text{ALL}_n g \Rightarrow \text{ALL}_n (f \circ g) \text{ et}$$

$$\text{ALL}_n f \circ (\text{ALL}_n g \circ h) \Rightarrow \text{ALL}_n (f \circ g) \circ h.$$

● On peut garder pour " $\text{ALL}_n$ " l'interprétation

$$I(\text{ALL}_n x) = n I(x) + 1$$

et on retrouve le même résultat pour ALLALL et ALLALL' dans les N-termes.

Par contre TLALL et TLALL'

$$\text{TLALL: } t1 \circ \text{ALL}_n f \Rightarrow \text{ALL}_{n-1} f \circ t1 \text{ SI (DEF } \circ \text{ALL}_n f)$$

$$\text{TLALL': } t1 \circ (\text{ALL}_n f \circ h) \Rightarrow \text{ALL}_{n-1} f \circ (t1 \circ h) \text{ SI (DEF } \circ (\text{ALL}_n f \circ h))$$

on a alors:

$$I(t1 \circ \text{ALL}_n f) = 2n I(f) + 2 > I(\text{ALL}_{n-1} f \circ t1) = 2(n-1) I(f) + 2$$

ainsi que

$$I(t1 \circ (\text{ALL}_n f \circ h)) = I(t1 \circ \text{ALL}_n f) I(h)$$

$$> I(\text{ALL}_{n-1} f \circ (t1 \circ h)) = I(\text{ALL}_{n-1} f \circ t1) I(h).$$

### REMARQUES

Nous prenons  $n$  égal à la longueur de la séquence plus 1 afin de garder des coefficients strictement positifs dans le cas de la séquence vide (ce qui permet de respecter la condition C1 sur l'interprétation des termes).

Tous les résultats restent valables avec  $n$  égal à la longueur de la séquence augmentée de  $k$  ( $k > 1$  quelconque).

#### 2.4. Règle TALL

s'écrit dans les N-termes:

$$\text{TALL: } T(x, x) \Rightarrow \text{ALL}_n [\text{id}, \text{id}] \circ x.$$

$$\text{On a } I(\text{ALL}_n [\text{id}, \text{id}] \circ x) = 4n I(x) + 1.$$

On est tenu d'interpréter " $T(x, y)$ " en tenant compte de la longueur commune des séquences données par  $x$  et  $y$ . Aussi, le morphisme  $\Delta$  substitue à " $T(x, y)$ " un des termes " $T_n(x, y)$ " où  $n$  est la longueur commune des séquences  $x$  et  $y$ . Les symboles " $T_n$ " sont ajoutés à l'ensemble des symboles des N-termes.

● L'interprétation de " $T_n(x, y)$ " est

$$I(T_n(x, y)) = 2n (I(x) + I(y)) + 2.$$

TALL s'écrit dans les N-termes:

$$T_n(x, x) \Rightarrow \text{ALL}_n [\text{id}, \text{id}] \circ x$$

$$\text{donc } I(T_n(x, x)) = 2n (I(x) + I(x)) + 2 = 4n I(x) + 2 > 4n I(x) + 1,$$

ce qui valide l'orientation de TALL.

La règle TRANS, introduite par le motif "trans  $\circ$   $[a, b]$ ", s'écrit dans les N-termes:

$$\text{TRANS: } T_n(x, y) \circ z \Rightarrow T_n(x \circ z, y \circ z).$$

● On complète l'ordre des symboles avec  $\alpha > T$  et on applique le cas 2.a de la définition de l'ordre.

#### 2.5. Règle entre "distl" et "append"

s'écrit dans les OP-termes:

$$\text{APPDISTL: } D(x, A(a, b)) \Rightarrow A(D(x, a), D(x, b)).$$

On ne peut prendre qu'une interprétation linéaire de  $D(x, a)$  à cause de la règle DISTL introduite par le motif "distl  $\circ$   $[x, a]$ ":

$$\text{DISTL: } D(x, a) \circ z \Rightarrow D(x \circ z, a \circ z)$$

L'interprétation du membre gauche de APPDISTL est alors supérieure à celle du membre droit. Il faut donc aussi distinguer les opérations  $D(x, a)$  selon la longueur de la séquence  $a$ . Le morphisme  $\Delta$  substitue à " $D(x, a)$ " un des termes " $D_n(x, a)$ " où  $n$  est la longueur de la séquence  $a$ . Les symboles " $D_n$ " sont ajoutés à l'ensemble des symboles des N-termes.

● L'interprétation de " $D_n(x, a)$ " est

$$I(D_n(x, a)) = n I(x) + I(a).$$

Dans les N-termes APPDISTL s'écrit:

$$D_{i+j}(x, A(a, b)) \Rightarrow A(D_i(x, a), D_j(x, b))$$

$$\text{et on a: } I(D_{i+j}(x, A(a, b))) = (i + j) I(x) + I(a) + I(b)$$

$$= I(A(D_i(x, a), D_j(x, b))) = i I(x) + I(a) + j I(x) + I(b).$$

Il suffit de compléter l'ordre sur les symboles avec  $D > A$  et d'appliquer le cas 2.a de la définition de l'ordre des termes.

Pour la règle DISTL

$$D_n(x, a) \circ z \Rightarrow D_n(x \circ z, a \circ z).$$

$$\begin{aligned} \text{On a : } I(D_n(x, a) \circ z) &= (n I(x) + I(a)) I(z) = n I(x) I(z) + I(a) I(z) \\ &= I(D_n(x \circ z, a \circ z)) = n I(x) I(z) + I(a) I(z). \end{aligned}$$

• On complète l'ordre sur les symboles avec  $\circ > D$  et on applique le cas 2.a de la définition de l'ordre des termes.

## 2.6. Autres règles des systèmes S-ALL

sont transformées dans les N-termes et n'apportent pas de difficulté particulière.

Nous donnons ici le détail de la preuve. Nous appellerons  $l$  et  $r$  les membres droits et gauches des règles.

### APPTRANS

$$\begin{aligned} T_{i+j}(A(a, b), A(c, d)) &\Rightarrow A(T_i(a, c), T_j(b, d)) \\ \text{SI } (DEF \circ [T_i(a, c), T_j(b, d)]) \end{aligned}$$

$$\begin{aligned} I(l) &= 2(i+j)(I(a) + I(b) + I(c) + I(d))2 \\ I(r) &= 2i(I(a) + I(c)) + 2 + 2j(I(b) + I(d)) + 2 \\ \text{On a } I(l) &> I(r) \text{ en remarquant que } 2j I(a) > 2. \end{aligned}$$

### ALLAPP

$$\begin{aligned} ALL_{i+j} f \circ A(a, b) &= A(ALL_i f \circ a, ALL_j f \circ b) \\ I(l) &= ((i+j) I(f) + 1)(I(a) + I(b)) \\ &= (i+j) I(f) I(a) + (i+j) I(f) I(b) + I(a) + I(b) \\ > I(r) &= (i I(f) + 1) I(a) + (j I(f) + 1) I(b) \\ &= i I(f) I(a) + I(a) + j I(f) I(b) + I(b). \end{aligned}$$

### ALLVID

$$ALL_1 f \circ <>^- = <>^-$$

$$I(l) = 2(I(f) + 1) = 2 I(f) + 2 > I(r) = 2.$$

### Les deux règles FALL

$$FALL: T_n(ALL_n f, ALL_n g) \Rightarrow ALL_n[f, g]$$

$$FALL': T_n(ALL_n f \circ h, ALL_n g \circ h) \Rightarrow ALL_n[f, g] \circ h.$$

Pour FALL, on a

$$\begin{aligned} I(l) &= 2n(n I(f) + 1 + n I(g) + 1) + 2 \\ &= 2n^2 I(f) + 2n^2 I(g) + 4n + 2 \end{aligned}$$

$$> I(r) = n(I(f) + I(g)) + 1 = n I(f) + n I(g) + 2.$$

Les résultats sont multipliés par  $I(h)$  pour FALL'.

### Les règles FALLg et FALLd

$$FALLg: T_n(id, ALL_n f) \Rightarrow ALL_n[id, f]$$

$$FALLd: T_n(ALL_n f, id) \Rightarrow ALL_n[f, id]$$

$$FALLg': T_n(h, ALL_n f \circ h) \Rightarrow ALL_n[id, f] \circ h$$

$$FALLd': T_n(ALL_n f \circ h, id) \Rightarrow ALL_n[f, id] \circ h$$

- Pour la règle FALLg

$$\begin{aligned} I(l) &= 2n(2 + n I(f) + 1) + 2 = 2n^2 I(f) + 6n + 2 \\ > I(r) &= n(2 + I(f)) + 1 = n I(f) + 2n + 1. \end{aligned}$$

- Pour la règle FALLd

$$\begin{aligned} I(l) &= 2n(I(h) + (n I(f) + 1) I(h)) + 2 \\ &= 2n^2 I(f) I(h) + 4n I(h) + 2 \end{aligned}$$

$$> I(r) = (n(2 + I(f)) + 1) I(h) = n I(f) I(h) + 2n I(h) + I(h).$$

- Les calculs sont les mêmes pour les règles FALLg' et FALLd'.

### Les règles ALLID, HALL, HALL', LALL, LALL' et TDISTL

ALLID:  $ALL_n id \Rightarrow id$

HALL:  $hd \circ ALL_n f \Rightarrow f \circ hd$  SI  $(DEF \circ ALL_n f)$

HALL':  $hd \circ (ALL_n f \circ h) \Rightarrow f \circ (hd \circ h)$  SI  $(DEF \circ (ALL_n f \circ h))$

LALL:  $last \circ (ALL_n f) \Rightarrow f \circ last$  SI  $(DEF \circ ALL_n f)$

LALL':  $last \circ (ALL_n f \circ h) \Rightarrow f \circ (last \circ h)$  SI  $(DEF \circ (ALL_n f \circ h))$

TDISTL:  $T_n (ALL_n ?l \circ D_n (x,a),a) \Rightarrow D_n (x,a)$

On a  $I(l) > I(r)$  pour chaque règle.

### ALLLIST

ALLLIST:  $ALL_n f \circ L(x) \Rightarrow f \circ x$

● On interprète

$$I(L(x)) = I(x) + 1$$

ce qui garantit la condition C2 sur l'interprétation.

On a alors

$$I(l) = (n I(f) + 1) (I(x) + 1) = n I(f) I(x) + n I(f) + n I(x) + 1$$

$$> I(r) = I(f) I(x).$$

Pour la règle LIST

$$L(x) \circ z \Rightarrow L(x \circ z),$$

$$\text{on a: } I(l) = I(x) I(z) + I(z) > I(r) = I(x) I(z) + 1.$$

Ce qui achève la preuve de terminaison des systèmes de réécriture S-ALL.

### 3. Systèmes de réécriture sur les primitives

Nous ne rencontrerons pas de difficulté supplémentaire. Nous comparons pour chaque règle les interprétations du membre gauche noté l et du membre droit noté r. Le lecteur peut sauter la lecture des calculs de ce paragraphe.

LTL:  $last \circ tl \Rightarrow last$  SI  $(DEF \circ (last \circ tl))$

$$I(l) = 4 > I(r) = 2.$$

LTL':  $last \circ (tl \circ h) \Rightarrow last \circ h$  SI  $(DEF \circ (last \circ (tl \circ h)))$

$$I(l) = 4 I(h) > I(r) = 2 I(h).$$

HLIST:  $hd \circ L(x) \Rightarrow x$  et LCVID:  $last \circ L(x) \Rightarrow x$

$$I(l) = 2 (I(x) + 1) > I(r) = I(x).$$

TLLIST:  $tl \circ L(x) \Rightarrow <>$  SI  $(DEF \circ x)$

$$I(l) = 2 (I(x) + 1) > I(r) = 2.$$

HTRANS et LTRANS:

$hd \circ T_n (a,b) \Rightarrow [hd \circ a, hd \circ b]$  SI  $(DEF \circ (hd \circ T_n (a,b)))$

$last \circ T_n (a,b) \Rightarrow [last \circ a, last \circ b]$  SI  $(DEF \circ (last \circ T_n (a,b)))$

$$I(l) = 2 (2n (I(a) + I(b)) + 2) = 4n I(a) + 4n I(b) + 4$$

$$> I(r) = 2 I(a) + 2 I(b).$$

TLTRANS:  $tl \circ T_n (a,b) \Rightarrow T_{n-1} (tl \circ a, tl \circ b)$

$$I(l) = 2 (2n (I(a) + I(b)) + 2) = 4n I(a) + 4n I(b) + 4$$

$$> I(r) = 2(n-1) (2 I(a) + 2 I(b)) + 2 = (4n-2) I(a) + (4n-2) I(b) + 2.$$

TLIST:  $T_1 (L(x), L(y)) \Rightarrow L([x, y])$

$$I(1) = 2 (1) (I(x) + 1 + I(y) + 1) + 2 = 2 I(x) + 2 I(y) + 6$$

$$> I(r) = I(x) + I(y) + 1.$$

HDISTL et LDISTL:

$$hd \circ D_n (x, a) \Rightarrow [x, hd \circ a]$$

$$last \circ D_n (x, a) \Rightarrow [x, last \circ a]$$

$$I(1) = 2 (n I(x) + a) = 2n I(x) + 2 I(a) > I(r) = I(x) + 2 I(a).$$

TLDISTL:  $tl \circ D_n (x, a) \Rightarrow D_{n-1} (x, tl \circ a)$

$$I(1) = 2 (n I(x) + I(a)) = 2n I(x) + 2 I(a)$$

$$> I(r) = (n-1) I(x) + 2 I(a).$$

DLIST:  $D_n (x, L(y)) \Rightarrow L([x, y])$

$$I(1) = I(x) + I(y) + 1 = I(r) = I(x) + I(y) + 1.$$

● On complète l'ordre sur les symboles avec  $D > L$  et on applique le cas 2.a de la définition de l'ordre des termes.

TVID:  $T_n (<>^-, <>^+) \Rightarrow <>^+$

$$I(1) = 2 (1) (2 + 2) + 2 = 10 > I(r) = 2.$$

DVID:  $D_1 (x, <>^+) \Rightarrow <>^+ SI (DEF \circ D_1 (x, <>^+))$

$$I(1) = (1) I(x) + 2 = I(x) + 2 > I(r) = 2$$

HIOTA et LIOTA:

$$hd \circ iota \Rightarrow 1^- SI (DEF \circ iota)$$

$$last \circ iota \Rightarrow id SI (DEF \circ iota)$$

$$I(1) = 4 > I(r) = 2.$$

HIOTA':  $hd \circ (iota \circ n) \Rightarrow 1^- SI (DEF \circ (iota \circ n))$

$$I(1) = 4 I(n) > I(r) = 2.$$

LIOTA':  $last \circ (iota \circ n) \Rightarrow n SI (DEF \circ (iota \circ n))$

$$I(1) = 4 I(n) > I(r) = I(n).$$

HAPP:  $hd \circ A(a, b) \Rightarrow hd \circ a$

SI (DEF  $\circ$  (hd  $\circ$  a))

$$I(1) = 2 (I(a) + I(b)) = 2 I(a) + 2 I(b) > I(r) = 2 I(a).$$

LAPP:  $last \circ A(a, b) \Rightarrow last \circ b$

SI (DEF  $\circ$  (last  $\circ$  b))

$$I(1) = 2 (I(a) + I(b)) = 2 I(a) + 2 I(b) > I(r) = 2 I(b)$$

TLAPP:  $tl \circ A(a, b) \Rightarrow A(tl \circ a, b)$

SI (DEF  $\circ$  (tl  $\circ$  a))

$$I(1) = 2 (I(a) + I(b)) = 2 I(a) + 2 I(b) > I(r) = 2 I(a) + I(b).$$

APPVID et APPVIDS:

$$A(a, <>^+) \Rightarrow a SI (DEF \circ A(x, <>^+))$$

$$A(<>^-, a) \Rightarrow a SI (DEF \circ A(x, <>^-))$$

$$I(1) = I(a) + 2 > I(r) = I(a).$$

Ce qui achève la preuve de terminaison des systèmes S-PRIMITIVE.

#### 4. Systèmes de réécriture S-\*\* et S-WHH

##### 4.1. Interprétation de "\*f"

Nous considérons la règle \*ALL:

$$*ALL: *f \circ [x, ALL \ g \circ a] \Rightarrow *(f \circ [?1, g \circ ?2]) \circ [x, a].$$

Nous ne trouvons pas d'interprétation de \*f qui nous permette d'avoir  $I(l) > I(r)$  (l et r notant les membres droits et gauche de \*ALL) compte tenu des interprétations de "x o y" et de "[x, y]".

Nous complétons  $\Phi$  qui remplace "\*f o [x, a]" par "\*\*(f, x, a)".

On ajoute la règle

$$*: *(f, x, a) \circ z \Rightarrow *(f, x \circ z, a \circ z)$$

qui assure la dualité des réécritures.

● On complète l'ordre des termes avec  $\circ > **$  et, si l'on interprète linéairement "\*\*", il suffit, pour la règle \*\* d'appliquer le cas 2.a de la définition de l'ordre des termes.

Dans les \*-termes, \*ALL devient:

$$*ALL: *(f, x, ALL_n \ g \circ a) \Rightarrow *(f \circ [?1, g \circ ?2], x, a).$$

● On interprète

$$I(**(f, x, a)) = I(x) + I(f) I(a).$$

On trouve alors pour \*ALL

$$I(l) = I(x) + I(f) (n I(g) + 1) I(a) = I(x) + n I(f) I(g) I(a) + I(f) I(a)$$

$$I(r) = I(x) + I(f) (2 + 2 I(g)) I(a) = I(x) + 2 I(f) I(g) I(a) + 2 I(f) I(a).$$

On peut sans changer les résultats des paragraphes précédents définir n comme étant la longueur de la séquence à laquelle s'applique "ALL<sub>n</sub> x" plus 2, c'est à dire  $n > 2$ .

Comme

$$I(f) I(g) I(a) > I(f) I(a), \text{ on peut alors conclure que } I(l) > I(r).$$

##### Les autres règles de S-\*\*

Ces règles ne posent pas de nouveaux problèmes. On peut sans inconvénient éviter de regarder le détail de la preuve qui est faite dans ce paragraphe.

\*ALL'

s'écrit dans les \*-termes:

$$*ALL': *(f, x, ALL_n \ g) \Rightarrow *(f \circ [?1, g \circ ?2], x, id)$$

$$I(l) = I(x) + I(f) (n I(g) + 1) = I(x) + n I(f) I(g) + I(f)$$

$$I(r) = I(x) + I(f) (2 + 2 I(g)) (2) = I(x) + 4 I(f) I(g) + 4 I(f)$$

Il faut encore augmenter n en prenant  $n > 4$ , et comme

$$I(f) I(g) > I(g), \text{ on peut conclure que } I(l) > I(r).$$

H\*

s'écrit dans les \*-termes:

$$hd \circ *(f, x, a) \Rightarrow x$$

et il est évident que l'on a  $I(1) > I(r)$ .

F\*

s'écrit dans les \*-termes:

$$T_n (**(f,x,a),**(f,y,a)) \Rightarrow **([f \circ [?1o?1,?2],g \circ [?2o?1,?2]], [x,y],a).$$

On a

$$\begin{aligned} I(1) &= 2n (I(x) + I(f) I(a)) + I(y) + I(g) I(a) + 2 \\ &= 2n I(x) + 2n I(y) + 2n I(f) I(a) + 2n I(g) I(a) + 2 \text{ et} \\ I(r) &= I(x) + I(y) + (6 I(f) + 6 I(g)) I(a) \\ &= I(x) + I(y) + 6 I(f) I(a) + 6 I(g) I(a). \end{aligned}$$

Comme  $n > 4$ , on a  $I(1) > I(r)$ .

Les règles F\*\* et F\*\*S

F\*\* s'écrit dans les \*-termes:

$$\begin{aligned} F**: T_n (**(f,x,t1 \circ **(g,y,z)),**(g,y,a)) \\ \Rightarrow **([f \circ [?1o?1,?2],g \circ [?2o?1,?2]], [x,y],a). \end{aligned}$$

On a

$$\begin{aligned} I(1) &= 2n (I(x) + I(f) (2) (I(y) + I(g) I(a)) + I(y) + I(g) I(a)) + 2 \\ &= 2n I(x) + 2n I(y) + 4n I(f) I(y) + 4n I(f) I(g) I(a) + 2n I(g) I(a) + 2 \\ \text{et } I(r) &= I(x) + I(y) + 4 I(f) I(a) + 6 I(f) I(g) I(a) + 6 I(g) I(a). \end{aligned}$$

Avec  $n > 4$ , on a  $4n I(f) I(g) I(a) > 4 I(f) I(a) + 6 I(f) I(g) I(a)$   
et  $2n I(g) I(a) \geq 6 I(g) I(a)$ , ce qui permet de dire que  $I(1) > I(r)$ .

Pour la règle symétrique F\*\*S, on trouve les mêmes interprétations des membres droits et gauches.

Les règles FALL\* et FALL\*S

FALL\* s'écrit dans les \*-termes:

$$\begin{aligned} \text{FALL*}: T_n (ALL_n g \circ **(f,x,a),**(f,x,a)) \\ \Rightarrow **([g \circ (f \circ [?2o?1,?2]),f \circ [?2o?1,?2]], [g \circ x,x],a). \end{aligned}$$

On trouve

$$\begin{aligned} I(1) &= 2n ( (n I(g) + 1) (I(x) + I(f) I(a)) + I(x) + I(f) I(a) ) + 2 \\ &= I(g) I(x) + I(x) + n I(g) I(f) I(a) + I(f) I(a) + I(x) + I(f) I(a) + 2 \\ &= 2n^2 I(g) I(x) + 4n I(x) + 2n^2 I(g) I(f) I(a) + 2n I(a) + 2 \\ \text{et } I(r) &= I(g) I(x) + I(x) + (6 I(g) I(f) + 6 I(f)) I(a) \\ &= I(g) I(x) + I(x) + 6 I(g) I(f) I(a) + 6 I(f) I(a). \end{aligned}$$

Avec  $n > 4$ , on peut conclure que  $I(1) > I(r)$ .

Pour la règle symétrique FALL\*S, on trouve les mêmes interprétations des membres droits et gauches.

Ce qui achève la preuve de terminaison du système de réécriture S-\*

#### 4.2. Interprétation de "WH(p,f)"

Nous complétons les \*-termes avec un symbole "W" tel que le morphisme  $\Xi$  transforme un terme "WH(p,f) o [x,a]" en "W(p,f,x,a)".

$\Xi$  introduit la règle:

$$W: W(p,f,x,a) \circ z \rightarrow W(p,f,x \circ z, z \circ z).$$

⊗ On interprète W par

$$I(W(p,f,x,a)) = I(x) + I(p) I(a) + I(f) I(a).$$

● On complète l'ordre des symboles par  $a > W$ .

Le calcul de l'interprétation des membres des règles de S-WHH est calqué sur celui des règles de S-\*\*: Il suffit d'ajouter la partie due à  $p$  qui est identique à celle de  $f$  pour un terme " $W(p, f, x, a)$ ".

On prouve ainsi la terminaison de S-WHH.

### 5. Systèmes de réécriture S-\* et S-WH

Considérons la règle \*LIST qui s'écrit dans les \*-termes:

\*LIST:  $W(f, x, L(y)) \Rightarrow A(L(x), L(f \circ [x, y]))$ .

Avec l'interprétation définie au paragraphe précédent, on trouve:

$I(1) = I(x) + I(f) (I(y) + 1) = I(x) + I(f) + I(f) I(y)$  et

$I(r) = I(x) + 1 + I(f) (I(x) + I(y)) + 1 = I(x) + I(f) I(x) + I(f) I(y) + 2$

On a donc  $I(r) > I(1)$  et la règle devrait être orientée dans l'autre sens.

Le même problème se pose aussi avec la règle \*CONS ainsi que, pour le système S-WH, avec les deux règles WHLIST2 et WHCONS.

Nous n'avons pas trouvé d'interprétation de  $W(f, x, a)$  qui convienne pour prouver la terminaison des systèmes de réécriture S-\* et S-WH. Nous n'avons pas non plus de contre exemple nous permettant de dire qu'ils ne se terminent pas. Nous pensons cependant que ces systèmes sont noethériens.

### Conclusion

A l'exception des systèmes S-\* et S-WH, nous avons pu faire la preuve

que les autres systèmes localement confluents S-BASE, S-PRIMITIVE, S-\*\*, S-WHH et S-COND que nous avons trouvés dans la première partie sont noethériens. La terminaison finie des systèmes S-\* et S-WH n'est pas prouvée car nous n'avons pas pu encore trouver une interprétation satisfaisante des termes " $f \circ [x, a]$ " et " $WH(p, f) \circ [x, a]$ ".

## References

- Ack79.W. B. Ackermann and J. B. Dennis, "VAL- A Value-Oriented Algorithmic Language: Preliminary Reference Manual," MIT/LCS/TR-218, 1979.
- Ars77.J. Arsic, La construction de programmes structurés, Dunod, Paris, 1977.
- Ash76.E.A. Ashcroft and W.W. Wadge, "Lucid - A Formal System for Writing And Proving Programs," Siam Journal on Computing, vol. 5, no. 3, 1976.
- Ash77.E.A. Ashcroft and W.W. Wadge, "Scope Structures And Defined Functions in Lucid," Theory of Computation Report 21, the University of Warwick, Oct 1977.
- Bac85.L. Bachmair and D. Plaisted, "Associative Path Orderings," 1st Conference on Rewriting Techniques and Applications, Dijon (FRANCE), 1985.
- Bac81a.  
John Backus, "The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, And Extended Definitions," Proceedings International Colloquium on the Formalization of Programming Concepts, Peniscola, Spain, pp. 1-43, Springer-Verlag LNCS 107, April 1981.
- Bac81b.  
John Backus, "Function Level Programs As Mathematical Objects," Proceedings of the Conference on Functional Programming Languages And Computer Architecture, Portsmouth, New Hampshire, pp. 1-10, Oct. 1981.
- Bac78.J. Backus, "Can Programming Be Liberated From the Von Neumann Style? A Functional Style And Its Algebra of Programs," Comm. of ACM, vol. 21, no. 8, pp. 613-641, 1978.
- Bar81.H. P. Barendregt, The Lambda Calculus. Its Syntax and Semantics, Studies in Logic and The foundation of Mathematics, 1981.
- Bau79a.  
F.L. Bauer and M. Broy, "Program Construction," in LNCS, vol. 69, Springer Verlag, Berlin, 1979.
- Bau79b.  
F. L. Bauer, M. Broy, N. Partsh, P. Pepper, and H. Wossner, "Systematics Transformation Rules," LNCS, vol. 69, Springer-Verlag, 1979.
- Bel84.F. Bellegarde, "Rewriting Systems on FP Expressions that Reduce the Number of Sequences they yield," in Symposium on LISP and Functional Programming, ACM, Austin, USA, 1984.
- Bel85.F. Bellegarde, "Rewriting Systems on FP Expressions to reduce the number of Sequences Yielded," Science of Computer Programming, vol. North Holland, 1985.
- Bel78.F. Bellegarde, J. P. Finance, B. Huc, J. Jaray, P. Lescanne, J. Marold, C. Pair, A. Quere, and J. L. Remy, "MEDEE: A Type of Language for the Deductive Programming Method," Conference on reliable software, German ACM Chapter, Bonn, 1978.
- Bel80.F. Bellegarde, A. Quere, and J.L. Remy, "Construction et Transformation Systématiques de Programmes," Rairo Serie Informatique, vol. 14,

pp. 219-252, 1980.

Ber81a.

G. Berry, "On the Definition of Lambda-Calculus Models," Proceedings International Colloquium on the Formalization of Programming Concepts, vol. 107, pp. 218-230, Springer-Verlag LNCS, Peniscola, Spain, April 1981.

Ber81b.

G. Berry, "Programming With Concrete Data Structures and Sequential Algorithms," Proceedings of the 1981 Acm Conference on Functional Programming Languages and Computer Architecture, pp. 49-57, Portsmouth, New Hampshire, Oct. 1981.

Bir35.G. Birkhoff, "On the Structure of Abstract Algebras," Proc. Cambridge Phil. Soc. 31, pp. 433-454, 1935.

Blo83.S. Bloom and R. Yindell, "Varieties of IF THEN ELSE," S.I.A.M. Journal of Computing, vol. 12, no. 4, pp. 677-707, 1983.

Bur75.W. H. Burge, Recursive Programming Techniques, The Systems Programming Series, 1975.

Bur77.R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," J of ACM, vol. 24, pp. 44-67, 1977.

Bur80.R.M. Burstall, D.B. MacQueen, and D.I. Sannella, "Hope: An Experimental Applicative Language," Report Csr-62-80, Department of Computer Science, University of Edinburgh, May 1980.

Car80.Luca. Cardelli, "Notes on the Vax Implementation of M1," Private Communication, Oct. to Dec. 1980.

Che83.P. Le Chenadec, "Formes Canoniques dans les Algèbres Finiment Présentées," Thèse de troisième cycle, Université de Paris-Sud, 1983.

Che85.A. Ben Cherifa, "Thèse de 3ème cycle," à paraître, 1985.

Cho85.C. Choppy and C. Johnen, "Petri nets: Proving Petri Net Properties With Rewriting Systems," RTA 1985, Dijon, 1985.

Cos85.S. Cosmadakis and P. C. Kanellakis, Two Applications of Equational Theories to Database Theory, Dijon, 1985.

Cou80.B. Courcelle and J.C. Raoult, "Completions of Ordered Magmas," Fundamenta Informaticae III.1, pp. 105-116, 1980.

Cou85.G. Cousineau, P. L. Curien, and M. Mauny, "The Categorical Abstract Machine," pub. LIIP, 1985.

Den81.Jack B. Dennis, "An Operational Semantics for A Language With Early Completion Data Structures," Proceedings International Colloquium on the Formalization of Programming Concepts, Peniscola, Spain. Springer-Verlag LNCS 107, pp. 260-267, April 1981.

Der82a.

Nachum Dershowitz, "Orderings for Term-Rewriting Systems," Theoretical Computer Science, vol. 17, pp. 279-301, 1982.

Der82b.

N. Dershowitz, Well-Founded Ordering, August 1982.

- Der83.N. Dershowitz, J. Hsiang, N.A. Josephson, and D.A. Plaisted, "Associative-Commutative rewriting," Proceedings of the 8th IJCAI, Karlsruhe (West Germany), pp. 940-944, 1983.
- Dij76.E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.
- Duc84.Amédée Ducrin, du problème à l'algorithme, Dunod, Paris, 1984.
- Fag83.F. Fages and G. Huet, "Unification and Matching in Equational Theories," Proceedings of CAAP 83, vol. 159, pp. 205-220, Springer Verlag, l'Aquila, Italy, 1983.
- Fes79.M. S. Feather, "A system for Developing Programs by Transformation," Ph. D Thesis, University of Edinburgh, 1979.
- Fin79.J.P. Finance, "Etude de la Construction Des Programmes: Méthodes Et Langages de Spécification et de Résolution de Problèmes," Thèse d'Etat, Université de Nancy I, 1979.
- Fin85.J.P. Finance and J. Souquières, "A method and a language for constructing iterative programs," Science of Computer Programming, North Holland, vol. 186, 1985.
- For84.R. Forgaard and J.V. Guttag, "REVE: A Term Rewriting System Generator with Failure-Resistant Knuth-Bendix," MIT-LCS, 1984.
- Fut85.K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer, "Principles of OBJ2," Proceedings, 12th ACM Symposium on Principles of Programming Languages Conference, 1985.
- Giv84.J. S. Givler and R. B. Kieburz, "Schema Recognition for Program

- Transformations," ACM Symposium on Lisp and Functional Programming, pp. 74-84, Austin, 1984.
- Glo80.P. Gloess and J.P. Laurent, "Adding Dynamic Paramodulation to Rewrite Algorithms," in Proceedings of the Fifth Conference on Automated Deduction, Lecture Notes in Computer Science, Springer Verlag, Les Arcs, France, July 1980.
- Gna85.I. Gnaedig and P. Lescanne, "Rewriting Systems for proving termination of rewriting systems," a paraître, 1985.
- Gog85.J. A. Goguen, J. P. Jouannaud, and J. Meseguer, "Operational Semantics for Order-Sorted Algebra," ICALP, Nafplion Greece, 1985.
- Gog79.J.A. Goguen and J.J. Tardo, "An Introduction to Obj, A Language for Writing And Testing Formal Algebraic Specifications," Specifications of Reliable Software Conference, Boston, 1979.
- Gor77.M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, "A Metalanguage for Interactive Proof in LCF," Internal Report Csr-16-77, Department of Computer Science University of Edinburgh, Sept. 1977.
- Gue77.I. Guessarian, "Les Tests et Leur Caractérisation Syntaxique," R.A.I.R.O. Informatique Théorique 11, pp. 133-156, 1977.
- Gui78.L. J. Guibas and D. K. Wyatt, "Compilation and Delayed Evaluation in APL," Proc. of 5th POPL, pp. 1-8, 1978.
- Gut81.John Guttag, James Horning, and John Williams, "FP With Data Abstraction And Strong Typing," Proceedings of the Conference on Functional Programming Languages And Computer Architecture, Portsmouth, New

- Hampshire, pp. 11-24, Oct. 1981.
- Guy84.J. Guyard and J. P. Jacquot, "MAIDAY : An environment for Guided Programming with a definitional language," Conf. Genie Logiciel, Orlando, 1984.
- Hen80.P. Henderson, Functional programming, Prentice-Hall International, London, 1980.
- Hen76.P. Henderson and H. Jr. Morris, "A Lazy Evaluator," Third Acm Conference on Principles of Programming Languages, pp. 95-103, 1976.
- Hoar72.C.A.R. Hoare, "Notes on Data Structuring," Structured Programming (O.J. Dahl E.W.Dijkstra C.A.R. Hoare), Academic Press, London And New York, pp. 83-174, 1972.
- Hsi81.J. Hsiang, "Refutational Theorem Proving Using Term Rewriting Systems," Ph.D Thesis, University of Illinois, Urbana, 1981.
- Hue78.G. Huet and B. Lang, "Proving And Applying Program Transformations Expressed With 2Nd Order Patterns," Acta Informatica 11, pp. 31-55, 1978.
- Hue80a.
- G. Huet, "Confluent reductions: abstract properties and applications to term rewriting systems," J. of ACM, vol. 27, no. 4, pp. 797-821, Oct. 1980.
- Hue81.G. Huet, "A complete proof of correctness of the Knuth-Bendix completion algorithm," JCSS 23,1, pp. 11-21, Aug. 1981.

- Hue80b.
- G. Huet and D. Oppen, "Equations and Rewrite Rules: A Survey," in Formal Languages: Perspectives And Open Problems, ed. Book R., Academic Press, 1980.
- Hul80.J.M. Hullot, "Compilation de Formes Canoniques dans les Théories Equationnelles," Thèse de 3ème Cycle, Université de Paris Sud, 1980.
- Jay83.B. Jayaraman and U. S. Reddy, "Theory of Linear Equations Applied to Program Transformation," IJCAI, pp. 10-16, Karlsruhe, 1983.
- Jen74.K. Jensen and N. Wirth, "Pascal User Manual And Report," Springer-Verlag, 1974.
- Jou82a.
- J.P. Jouannaud, C. Kirchner, and H. Kirchner, "Incremental Unification in Equational Theories," Proc. of the Allerton Conference, 1982.
- Jou83.J. P. Jouannaud, C. Kirchner, and H. Kirchner, "Incremental Construction of Unification Algorithms in Equational Theories," in Proceedings of the International Conference On Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 154, pp. 361-373, Springer Verlag, Barcelona Spain, 1983.
- Jou84a.
- J.P. Jouannaud and H. Kirchner, "Completion of a set of rules modulo a set of equations," Proceedings 11th ACM Conference of Principles of Programming Languages, Salt Lake City (Utah, USA), 1984.
- Jou81a.

J.P. Jouannaud and Y. Kodratoff, "Program Synthesis From Example of Behaviour," Proc. of the International Workshop on Program Construction. Chateau De Bonas. Ed. Biermann And Guiho. Reidel Publish, 1981.

Jou82b.

J.P. Jouannaud, P. Lescanne, and F. Reinig, "Recursive Decomposition Ordering," in Formal Description of Programming Concepts 2, ed. Bjørner D., pp. 331-346, North Holland, Garmish Partenkirchen, RFA, 1982.

Jou84b.

J.P. Jouannaud and M. Munoz, "Termination of a set of rules modulo a set of equations," Proceedings 7th Conference on Automated Deduction, vol. 170, Napa Valley (California, USA), 1984.

Jou81b.

J.P. Jouannaud and F. Reinig, "Decomposition Orderings: A New Family of Recursive Simplification Orderings," t.f des s.r, crin, 1981.

Kep84.Stephane Kaplen, "Fair Conditional Term Rewriting Systems: Unification, Termination and Confluence," Une étude des systèmes de réécriture conditionnelle., Orsay, 1984.

Kie81.Richard B. Kieburtz, "Transformations of FP Program Schemes," Proceedings of the Conference on Functional Programming Languages And Computer Architecture, Portsmouth, New Hampshire, pp. 41-48, Oct. 1981.

Kir85b.

C. Kirchner, "Méthodes et outils de conception systématique

d'algorithmes d'unification dans les théories équationnelles," Thèse de doctorat d'Etat, Université de Nancy I, 1985.

Kir82.C. Kirchner and H. Kirchner, "Contribution à la résolution d'équations dans les algèbres libres et les variétés équationnelles d'algèbres," Thèse de 3ème cycle, Université de Nancy I, 1982.

Kir84.C. Kirchner and H. Kirchner, "Implementation of a General Completion Procedure Parameterized by Built-in Theories And Strategies," Rapport Crin 84-R-85, 1984.

Kir85a.

H. Kirchner, "Preuves par complétion dans les variétés d'algèbres," Thèse de doctorat d'Etat, Université de Nancy I, 1985.

Knu70.D. Knuth and P. Bendix, "Simple Word Problems in Universal Algebras," Computational Problems in Abstract Algebra Ed. Leech J., Pergamon Press, pp. 263-297, 1970.

Lan66.P. J. Landin, "The Next 700 Programming Languages," Communications of the Acm, Vol. 9, No 3, March 1966.

Lan79.D.S. Lankford, "On Proving Term Rewriting Systems Are Noetherian," Report Mtp-3, Math. Dept., Louisiana Tech University, May 1979.

Les81.P. Lescanne, "Decomposition Orderings As A Tool to Prove the Termination of Rewriting Systems," Proc. 7th Intl Joint Conf on Artificial Intelligence, pp. 548-550, vancouver canada, Aug. 1981.

Les83.P. Lescanne, "Computer Experiments with the REVE Term Rewriting System Generator," in 10th ACM Conf. on Principles of Programming

Languages, pp. 99-108, Austin Texas, January 1983.

Les84a.

P. Lescanne, "Uniform termination of term rewriting systems - Recursive decomposition ordering with status," Proceedings 9th Colloques les Arbres en Algebre et en Programmation, Bordeaux (France), 1984.

Les84b.

P. Lescanne, "Term rewriting systems and algebra," Proceedings 7th international Conference on Automated Deduction, Napa Valley (California, USA), 1984.

Les84c.

P. Lescanne and H. Zhang, "extended path of symbols ordering (abstract)," Proceedings of the 5th ECAI, Pisa, Italy, 1984.

Man70.Z. Manna and S. Ness, "On the Termination of Markov Algorithms," Third Hawaii International Conference on System Sciences, pp. 789-792, 1970.

Man73.Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods in Proving Properties of Programs," Comm. ACM, vol. 16, no. 8, pp. 491-502, 1973.

McC62. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, "Lisp 1.5 Programmer'S Manual," the M.I.T. Press, 1962.

Niv75.M. Nivat, "On the Interpretation of Recursive Polyadic Program Schemes," Symposia Mathematica Vol. Xv, Istituto Nazionale Di Alta Matematica Italy, pp. 225-281, 1975.

Mil78. Robin Milner, "A Theory of Type Polymorphism in Programming," Jcss 17, pp. 348-375, 1978.

Pai79.C. Pair, "La construction des Programmes," RAIRO Informatique, vol. 13, no. 2, pp. 113-118, 1979.

Pet81.G. Peterson and M. Stickel, "Complete sets of reduction for equational theories with complete unification algorithms," J. of ACM, vol. 28, no. 2, pp. 233-264, 1981.

Pla78.D. Plaisted, "A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems," Dept. of Computer Science Report 78-943, U. of Illinois At Urbana-Champaign, Sept. 1978.

Red82.U. S. Reddy, "Programming with Sequences," ACM Southeast Regional Conf., 1982.

Rem82.J.L. Remy, "Etude de la Réécriture Conditionnelle et de Ses Applications à la Spécification, à l'Implémentation et à la Certification des Types Abstraits Algébriques," Thèse d'Etat, Université Nancy 1, 1982.

Rem84.J.L. Remy and H. Zhang, "REVEUR 4: a System for Validating Conditional Algebraic Specifications of Abstract Data Types," Proceedings of the 5th ECAI, Pisa, 1984.

Sou85.J. Souquieres and J.P. Finance, "Description and improvement of iterative program transformations," Science Computer Programming. North Holland, 1985.

Tur76.D. A. Turner, "SASL Language Manual," St Andrews University, Technical Report, 1976.

Tur81.D.A. Turner, "The Semantic Elegance of Applicative Languages,"  
Proceedings of the Conference on Functional Programming Languages And  
Computer Architecture, Portsmouth, New Hampshire, pp. 85-92, Oct. 1981.

Wad81.P. Wadler, "Applicative Style Programming, Program Transformation,  
 And List Operators," Proceedings of the Conference on Functional Pro-  
gramming Languages And Computer Architecture, Portsmouth, New  
Hampshire, pp. 25-32, Oct. 1981.

Wad84.P. Wadler, "Listlessness is Better than Laziness: Lazy Evaluation and  
 Garbage Collection at Compile Time," ACM Symposium on Lisp and Func-  
tional Programming, pp. 45-52, Austin, 1984.

Weg76.B. Wegbreit, "Goal Directed Program Transformation," IEEE Trans. on  
Software Engineering, vol. 2, pp. 69-80, 1976.

Wil80.J. H/ Williams, "On the Development of the Algebra of Functional Pro-  
 gramming," Report RJ2983 IBM Research Laboratory, San Jose, 1980.

Wir73.N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall  
 series in Automatic Computation, 1973.

NOM DE L'ETUDIANT : Madame BELLEGARDE Françoise

NATURE DE LA THESE : Doctorat d'Etat ès sciences

VU, APPROUVE ET PERMIS D'IMPRIMER

NANCY, le 20 JUIN 1985 n° 1084

LE PRESIDENT DE L'UNIVERSITE DE NANCY I



Quand on construit un programme itératif à l'aide d'un raisonnement structuré, on introduit des séquences (ou suites) intermédiaires inutiles. Nous voulons définir un système de transformation permettant de les supprimer.

Nous codons les programmes dans un langage fonctionnel, le langage FP, qui nous permet de voir les programmes d'un point de vue algébrique. Nous utilisons des fonctionnelles pour coder les itérations.

L'ensemble A des axiomes de la FP-algèbre comprend les axiomes qui concernent toutes les manières de composer les fonctions ayant pour argument et pour résultat des séquences. Ainsi chaque classe de A-égalité contient un terme qui n'engendre pas de séquence intermédiaire inutile. Nous orientons ces axiomes en règles d'un système de réécriture de manière à ce que la relation de réduction supprime les séquences intermédiaires inutiles et la forme normale d'un FP-terme, si elle existe, n'engendre plus de séquence intermédiaire inutile. L'idéal est que tout terme ait une forme normale unique. Mais il faut pour cela que le système de réécriture obtenu soit convergent (noethérien et confluent). La procédure de Knuth-Bendix permet de transformer un ensemble d'équations en un système de réécriture convergent. Nous avons utilisé la procédure de Knuth-Bendix implémentée dans le système REVE et obtenu un système de réécriture convergent. REVE nous donne alors la forme normale  $t'$  d'un FP-terme  $t$  pour ce système de réécriture.  $t'$  génère moins de séquences intermédiaires que  $t$ . Notre système de réécriture est capable d'éliminer des séquences intermédiaires dues aux compositions des fonctions qui codent les itérations.

Une première partie de la thèse est consacrée à l'exposé des résultats acquis en utilisant la version de REVE dite 2.2. Une deuxième partie permet d'envisager un plan de ce que l'on pourra réaliser dans l'avenir avec les versions ultérieures de REVE.